Compilers And Code Generation

Elizabeth Scott

CS3470

Department of Computer Science
Egham, Surrey TW20 0EX, England

Abstract

These lecture notes accompany the course CS3470 Compilers And Code Generation. They contain the basic material covered in that course.

This document is © Elizabeth Scott and Adrian Johnstone 1998, 2008, 2011, 2013.

Permission is given to freely distribute this document electronically and on paper. You may not change this document or incorporate parts of it in other documents: it must be distributed intact.

Please send errata to the authors at the address on the title page or electronically to `E.Scott@rhul.ac.uk`.

Contents

1 Languages and translation

In this first section we shall discuss the general principles of language and translation which underpin the theory and implementation of compilers. We shall also give an overview of the course structure, objectives, and administrative organisation.

This section only contains a brief outline of the motivation and historical development of compiler theory. You should read one of the texts recommended below to fill in the details in these more sparse notes.

1.1 High level and low level languages

Broadly speaking, a high level language provides a notation that is designed to map easily onto human design procedures and a low level language is designed to map easily onto machine operations. A language construct that generates many machine actions (such as a FOR loop) is called high level, and a language construct that generates only one action (such as assignment) is called low level.

In the past, low level languages (those which contain no high level constructs at all) were used by programmers seeking the most efficient programs, because all aspects of the program's execution can be explicitly specified. In a high level language, the compiler makes some implementation decisions for us. High level languages can still allow a degree of access to potentially dangerous machine capabilities, such as direct manipulation of addresses.

Although a low level language presents real opportunities for program optimisation, they are hard to write and usually non-portable, because every computer architecture has its own machine instructions and thus its own assembly language. Thus programs are usually written in high level languages and automatically translated into the appropriate machine language.

1.2 Course organisation

Aims:

- ◇ To develop the theory that makes it practicable to produce efficient compilers.
- ◇ To describe several different methods of automatic language analysis, discussing their advantages and disadvantages.
- ◇ To explain how to design computer languages so that efficient compilers can be written for them.
- ◇ To give insight as to why programming languages look the way they do.

Learning objectives:
By the end of this course a student should be able to

- ◇ explain the role and structure of a compiler and be able to describe the standard stages of compilation

- ◇ write grammars to specify simple languages, and write a lexical analyser and parser for them
- ◇ describe and write syntax directed translators and use them to construct intermediate code
- ◇ describe standard techniques for improving target code quality

There will usually be three lectures a week. One lecture slot will be sometimes be run as an interactive class with an associated work sheet.

There will be four assessed assignments, each worth 5% of the course mark.

This set of departmental lectures notes accompanies the lectures, and can be found on the course website. The material in these notes will be expanded on in class, so **you will need to take your own notes to supplement the course notes**. A half unit course represents an average student workload of 9 hours a week, so you should expect to spend about six hours a week outside lectures studying for this course (you may wish to spend longer if you find the material particularly difficult or if you hope to get a top class degree). We recommend that you use the notes provided as a basis for studying text books, and trying the exercises that they contain. Recommended books for this course are:

- ◇ *Compilers: principles, techniques and tools*
  Alfred V. Aho, Monica S Lam, Ravi Sethi and Jeffrey D. Ullman, Addison-Wesley, 2007.
- ◇ *The theory and practice of compiler writing* J. Tremblay and P. G. Sorenson, McGraw Hill 1985 (dense but comprehensive).

You may also find useful help and information on the net, particularly via the news group `comp.compilers`.

1.3 Related topics and transferable skills

1.3.1 Bringing together earlier topics

This course applies many of the ideas covered in the first year Machine Fundamentals course, particularly automata theory and assembly programming. It also applies the basic mathematical skills developed in the first year Mathematical Structures course.

We will also apply the knowledge you have built up of various programming languages. This experience will help you to understand the purpose and usefulness of the techniques that we will study,

1.3.2 Applying the ideas gained

This course course includes laboratory sessions using `rdp`, our recursive descent parser generator.

The `rdp` tool uses the approaches covered in this course to generate parses for user specified grammars. You will be able to apply your understanding of

the LL(1) grammar conditions to write language specifications which can be accepted by `rdp`.

You will also be able to add semantic rules to grammars so that a corresponding `rdp` parser can actually translate input programs, either into assembly language or into a directly executable *C* program.

1.3.3 Personal development

As well as academic knowledge, our degree programmes build transferable skills. Throughout this course you will need to work through exercises, both following examples presented in lectures and working on you own. This will reinforce the general skills of

- ◇ practising in order to gain mastery of a technique
- ◇ using concrete examples to build understanding of general principles

The inclusion of lab sessions to reinforce the theoretical material develops the skill of

- ◇ using different approaches to provide alternative methods for understanding

In general, the regular assignment sheets will extend your transferable skills in

- ◇ problem solving
- ◇ personal organisation
- ◇ working to deadlines
- ◇ ability to communicate understanding and knowledge

1.4 Translation

When translating we usually begin by detecting individual words and then we check the relationships that allows them to be grouped into phrases. Finally we select a meaning for the phrase. Both human and computer languages exhibit three main features:

1. vocabulary, the basic words from which phrases are constructed,
2. grammar, the rules by which words may be combined to form phrases,
3. semantics, the meaning which may be extracted from correctly formed phrases.

1.5 Vocabulary

The vocabulary of computer languages is very small compared to that of human languages, the bulk of most programs being made up of *names* that are defined during program translation. Mathematical symbols also count as words in programming languages, so that the following character strings all represent discrete words to a Pascal translator:

`begin` `end` `for` `while` `+`
`mod` `:=` `>=` `;` `if` `integer`
`1001.3` `variable_name`

1.6 Semantics

Of the three elements of language, the semantics of a language are the most difficult to characterise in any formal sense. Translation essentially involves the construction of a string in the object language which has the same meaning (semantics) as a given string in the source language.

Natural language translation is not an easy process because of the context sensitivity. It is not possible to translate 'word by word' or even to correctly translate individual words in isolation. The French phrase *un homme at un enfant fatigué* translates as *a man and a tired child*, but a literal word-for-word translation yields *a man and a child tired* which is not acceptable English. More interestingly, the almost identical *un homme at un enfant fatigués* means a tired man and a tired child even though the word for word translation would be as above.

Furthermore, if we only see part of a sentence we may not be able to tell what all the words mean. For example the phrase '... giant waves down the tunnel ....' is part of the following two sentences.

The giant waves down the tunnel to the child waiting at the other end.

The rushing water came in giant waves down the tunnel.

The meaning of the word 'waves' is different in each case, and can only be established by examining the surrounding context. In order to allow efficient compilation techniques we usually require that computer languages do not have context sensitivities.

1.7 Grammar

The problem of the limitation of word-for-word substitution holds for computer languages as well as for natural languages. A simple example is the structure of primitive data declarations in Pascal and C. In C, a declaration is introduced by a type name and followed by a comma-delimited list of new names, one *per* new variable:

```
int a,b,c;
```

In Pascal, a block of variable declarations is introduced by the keyword `VAR` followed by a comma delimited list of new names, followed by a colon (`:`) and the name of the variable type.

```
VAR a,b,c:  integer;
```

If we wanted to write a translator from Pascal to C then we would need to read the entire data declaration phrase and reorder it before emitting the Pascal equivalent. Most Unix systems come with a program `p2c` which performs exactly this task – it translates correct Pascal programs to equivalent C programs. Figures 1 and 2 show an example of this process.

```
{ Pascal source code }
procedure lookup(var name : str255; var np : nodeptr);
   var
      npp : ^nodeptr;
   begin
      if strlen(name) > maxnamelen then
         setstrlen(name, maxnamelen);
      npp := addr(base);
      while (npp^ <> nil) and (npp^^.name <> name) do
         begin
            if name < npp^^.name then
               npp := addr(npp^^.left)
            else
               npp := addr(npp^^.right);
         end;
   end;
```

Figure 1 p2c example Pascal source code

```
/* C output code */
Static Void lookup(name, np)
Char *name;
node **np;
{
  node **npp;

  if (strlen(name) > maxnamelen)
    name[maxnamelen] = '\0';
  npp = &base;
  while (*npp != NULL && strcmp((*npp)->name, name)) {
    if (strcmp(name, (*npp)->name) < 0)
      npp = &(*npp)->left;
    else
      npp = &(*npp)->right;
  }
}
```

Figure 2 p2c resulting C code

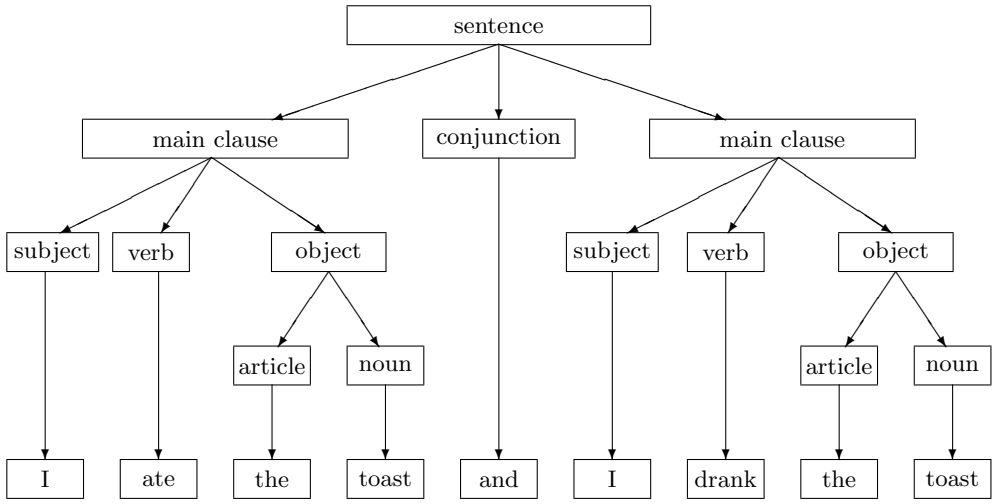


Figure 3 The structure of an English sentence

We use grammar rules to define the correct structure of a particular language construct. When learning human languages, grammar rules are usually presented rather informally, for example

> 'Though nouns of multitude may be freely used with either a singular or a plural verb, or be referred to by pronouns of singular or plural meaning, they should not have both (except for special reasons and upon deliberation) in the same sentence; and words that will rank in one context as nouns of multitude may be very awkward if so used in another'.

This rather confusing paragraph attempts to catch detailed nuances of meaning. At a coarser level of analysis, tree diagrams are very useful in establishing the relationship between the grammatical elements. Figure 3 shows a five level hierarchy that demonstrates the structure of two clauses and their constituent parts. Exactly the same kind of diagram can be drawn to show the relationship between the elements of a Pascal program fragment.

Tree diagrams for complete computer languages are extremely unwieldy. A *generative grammar* is a notation for expressing the relationships in the tree diagram in a textual fashion. The notation we use is the Backus-Naur Form (BNF) originally developed in order to define the syntax of Algol-58 and based on the ideas of Noam Chomsky who in the 1950's revolutionised the linguist's approach to grammatical analysis.

The theoretical study of language translation has led to the design of computer languages which can be translated in a reasonably straight-forward way – in fact, by a computer. This is a real success story of CS as a science. An 'experiment' was performed (a language and compiler were written by hand), then the results were analysed leading to theoretical understanding of the prob-

lems in compilation, and then this theory was fed back into language design in order to make compiler writing easier.

The first part of this course will develop the language theory which underpins automatic translation. In order to discuss the theory of language translation we need to have a formal definition of a language. You have studied formal definitions regular of languages in CS1870. We shall build on the ideas developed in that course. You will probably need to remind yourself of the CS1870 material. You can also find everything you need in the Aho, Lam, Sethi and Ullman book which is recommended for this course. In Section 4.3 there is a set of exercises which you are strongly encouraged to complete.

1.8 Course outline

1. Language and translation
2. Stages of compilation
3. Lexical analysis and symbol tables
4. Recursive descent parsers
5. Table based parsers
6. General parsers
7. Semantic attributes and intermediate code
8. Control and data flow analysis
9. Code improvement for sequential processors

1.9 Professional issues

As computer science professionals we must act according to the ethical codes of the profession. Both the ACM/IEEE and the BCS have codes of conduct and good practice. In this section we shall highlight aspects professional conduct that relate particularly to the design and use of compilers and translators.

The ACM's Code of Ethics and Professional practice has a short form, reproduced below, and a full form that you should read on their website, `www.acm.org/about/se-code`.

ACM/IEEE Code of Ethics Short Form
Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. PUBLIC - Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public

interest.
3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.
8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

There is a **BCS Code of Conduct**, which can be found on their website: `//www.bcs.org/category/6030`.

In Section 2(b) is listed the need to maintain your professional knowledge, skills and competence, maintaining awareness of technological developments, procedures, and standards that are relevant to your field.

The strong foundational knowledge that is obtained from an academic computer science degree forms a basis for this and for items 3,4 and 8 of the ACM Code above.

Annex A, Interpretation of the BCS Code of Conduct, of the BCS code gives the following details which emphasise the need for technical competence. This is also reflected in Principles 3 and 8 of the ACM

BCS: Professional Competence and Integrity

• All members are required to undertake professional development activities as a condition of membership. Continuing professional development activities should broaden your knowledge of the IT profession and maintain your competence in your area of specialism.

• You should seek out and observe good practice exemplified by rules, standards, conventions or protocols that are relevant in your area of specialism.

• You should only claim current competence where you can demonstrate you have the required expertise e.g. through recognised competencies, qualifications or experience.

ACM Principle 3: PRODUCT
Software engineers shall, as appropriate:
3.01. Strive for high quality, acceptable cost and a reasonable schedule, ensuring significant tradeoffs are clear to and accepted by the employer and the client, and are available for consideration by the user and the public.
3.02. Ensure proper and achievable goals and objectives for any project on which they work or propose.
3.03. Identify, define and address ethical, economic, cultural, legal and environ-

mental issues related to work projects.
3.04. Ensure that they are qualified for any project on which they work or propose to work by an appropriate combination of education and training, and experience.
3.05. Ensure an appropriate method is used for any project on which they work or propose to work.
3.06. Work to follow professional standards, when available, that are most appropriate for the task at hand, departing from these only when ethically or technically justified.
3.07. Strive to fully understand the specifications for software on which they work.
3.08. Ensure that specifications for software on which they work have been well documented, satisfy the users requirements and have the appropriate approvals.

ACM Principle 8: SELF
Software engineers shall continually endeavor to:
8.01. Further their knowledge of developments in the analysis, specification, design, development, maintenance and testing of software and related documents, together with the management of the development process.
8.02. Improve their ability to create safe, reliable, and useful quality software at reasonable cost and within a reasonable time.

A substantial component of the cost of a software engineering project is the time taken for software development. As is made explicit in ACM Principle 3.01 and 8.02, it is the professional responsibility of a software developer to use methods which allow the time taken to be minimised and their software to be correct and efficient.

Throughout the 1950s compilers were considered to be difficult to write. The first Fortran compiler took 18 staff years to implement. In the area of language translation there has been particular theoretical success, the study of systematic ways of describing the structures and meanings of computer languages has resulted in a theoretical base which allows a competent programmer to produce a compiler for a complex language in only a few weeks. It is also the case that the knowledge of this theory has had a great influence on the 'shape' of modern computer languages; they are designed to allow the application of standard compiler-generation techniques. This accounts for many of the differences between languages such as COBOL and Java.

If the theory that has been developed is not part of a software writer's armoury then they are likely to design languages for which translators are expensive to construct and maintain. Since almost all forms of interface with a computer require, often multiple levels of, translation the impact of this could be substantial.

The material covered in this course will underpin high standards of computer language and compiler design. It will also provide principles against which designs for new languages can be assessed for ease of implementation.

It is the responsibility of all software builders to produce correct code, and

it is particularly important that a compiler is correct since it generates the program that is actually executed. Automatically generated programs are more likely to be error free as the generator program is used frequently and as long as it is correct the programs it generates will be correct. The emphasis in this course is not 'can you do it' but 'can you write a program to do it'. Thus automatic compiler generation is desirable not only for efficiency but also for correctness. This means that it is necessary to master the systematic approach and detailed methods presented in this course, rather than producing ad hoc solutions. These methods form the basis of compiler-compilers, programs which generate compilers. In particular, we will study methods to automatically generate lexical analysers from sets of regular expression, and parsers from classes of context free grammars.

The material in this course forms part of your professional armoury.

2 Stages of compilation

Computer language translation is traditionally viewed as a process with two main parts: the *front end* conversion of a high level language text into an intermediate form, and the *back end* conversion of the intermediate form into the native language of a computer. This approach is useful because it turns out that the challenges encountered in the design of a front end differ fundamentally from the problems posed by back end code generation and separating out the problems makes it easier to think about the overall task.

A pure translation program that converts from one language such as C to another, such as the native machine language of your computer is called a *compiler* because it compiles a list of machine-level instructions from a high level specification that is independent of the type of computer you are using.

The language to be translated forms the input to the front end and is called the *source* language. The output of the back end is called the *target* or *object* language.

2.1 Interpreters

Sometimes the subdivision of the translation problem into front and back ends is explicit in the translator program, but not always. An *interpreter* is a special kind of language translator that executes actions as it translates. Most operating system command shells are of this form: each command is executed as it is encountered. In such a system there is no readily discernible back end or intermediate form although it can still be useful to think of the program as performing front and back end tasks. The macro languages found in most word processors, along with simple programming languages such as BASIC are most often implemented as interpreters.

2.2 Portability

In systems that do maintain a division between front and back ends, it is in principle possible to use a single intermediate form with multiple front and back ends as illustrated in Figure 4.

The intermediate form must provide enough generality to cope with the various source and target languages. Fortunately, front end processors for different languages sometimes display striking similarities. For instance, at a very crude level the variable declaration constructs in C and Pascal are quite similar. Their use of IF-THEN-ELSE selection is almost identical. It is perfectly possible to design an intermediate form that can cope with both C- and Pascal-like structures.

Using this organisation, a compiler for a given language can be moved to a new computer architecture by writing a new back end to take account of the differing instruction sets. More rarely, a new programming language syntax can be quickly implemented on a given architecture by building a new front end and using an existing back end. This saving in engineering effort can be very important in commercial compiler systems, even though it may require

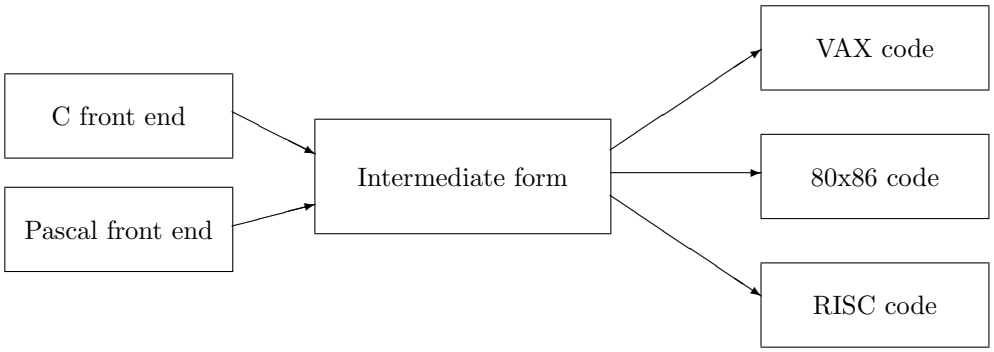


Figure 4 Multiple front and back ends

an intermediate form that is more complex than that required for a single source/target language pair.

2.3 Automated front end production

Many of the theoretical issues surrounding front end translation were solved during the 1960's and 1970's, and it is possible to reduce most of the implementation effort for a new front end to a clerical exercise that may itself be turned into a computer program. *Compiler-compilers* are programs that take the description of a programming language usually written in some variant of BNF, and output the source code of a program that will recognise, and possibly act upon, phrases written in that language.

2.4 Back end design

Code generation, the primary task of the back end, is much less well understood than front end translation. The basic task is the selection of machine code sequences that correctly represent the meaning of the source language phrases. In general we will want to generate code which executes either as quickly as possible, or requires as little space as possible, or both (these two aims may or may not conflict).

2.5 How a computer translates

We have already seen in section 2.1 that translators may be conveniently described in terms of a front end, an intermediate form and a back end. In more detail, many (although not all) compilers display the structure shown in Figure 5. Each of the seven boxes represents a compiler *phase*, and might be thought of as an independent program module, with modules passing information along in pipeline fashion.

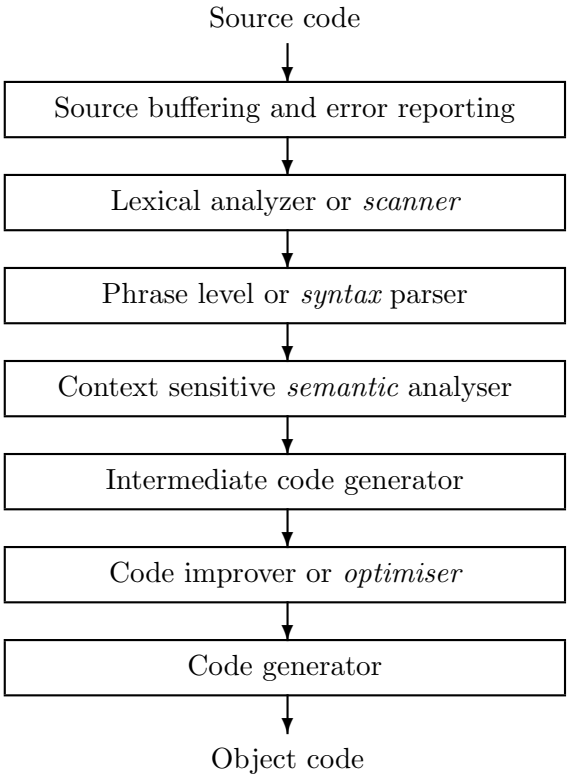


Figure 5 The traditional compiler phases

2.5.1 Source buffering and error reporting

For efficiency an entire line of source code is read from disk, and character by character analysis proceeds in memory. This module usually also handles the generation of source code listings, and the insertion of error messages.

2.5.2 Lexical analyser

The lexical analyser breaks the input stream of characters up into a series of language tokens. In a computer language, the 'words' might include punctuation symbols as well as keywords.

Since the vocabulary of programming languages is so small, it is convenient to allocate a small integer to each keyword and use this numeric label in later stages of processing rather than the full text string. The output of the lexical analyser, then, is a stream of integers representing the stream of tokens recognised within the stream of characters coming from the source buffer.

The lexical analyser also discards formatting information, such as whitespace, newlines and comments, that is used purely to help human understanding of the source program.

2.5.3 Syntax analysis – parsing

A *parser* is a program that checks that a string conforms to the grammar rules for a language. The grammar rules are usually represented by BNF productions, and often a *parser generator* is used to automate the essentially clerical process of constructing a parser for a given set of productions.

We are interested mainly in translations that can occur in near *linear* time, so that translation time is proportional to the number of words in the text that is to be translated. To make this possible, programming languages are designed to be unambiguous and in fact the meaning of most computer language phrases is not only clear in isolation, but may be built up in a single left to right scan of the program text.

It is not straightforward to discover if a particular set of grammar productions may be parsed in linear time, not least because there are several known linear-time parsing algorithms, each of which imposes different constraints on the kind of grammar productions that may be written.

2.5.4 Semantic analyser

At this stage, using the structure determined in the syntactical analysis, the meaning of the input is determined. To describe meaning we give it in a language that we already understand. i.e. German to English to French. For compilers this is often done by producing 'intermediate code'.

2.5.5 Intermediate code generator

There are several popular types of intermediate form, including the use of a pseudo-assembly language for an idealised machine or the construction of

derivation trees which are effectively a map of the grammar productions recognised by the parser. A good intermediate form will be able to encode the concepts used in several different languages.

2.5.6 Code improver

It is not uncommon for programmers to write code that contains redundancies such as the following pair of array accesses.

```
a[i*j] = p;
b[i*j] = q;
```

A more efficient version of the same program fragment is:

```
temp = i*j;
a[temp] = p;
b[temp] = q;
```

Here, the indexing expression has been pre-calculated, saving one multiplication.

It is possible to write a program that traverses the intermediate form and automatically rearranges the original code in this way, a process known as *common sub-expression elimination*. In fact this is only one of a family of techniques that has been developed to improve user code by reordering and sometimes completely replacing the users instructions with equivalent code that is faster (or smaller) *whilst preserving program semantics*. Traditionally compilers employing these techniques are called 'optimising' compilers which is perhaps a misnomer since in general the truly optimal code sequence is not produced. We prefer to use the term 'code improvement' to 'code optimisation'.

2.5.7 Code generator

Code generators traverse the intermediate form and output actual fragments of the target language which implement the specified semantics. Code generators are sometimes based on pattern matchers, that look for particular configurations in the intermediate form and check them against a list of stored code templates. Perhaps the most important sub-function of the code generator is to perform *register allocation* in which the most frequently used variables are assigned to machine registers as opposed to being held in the slower main memory.

The output of the code generator may be source for an assembler, or the equivalent binary object code. Some compilers have an option to produce assembly language even though, for efficiency reasons, they usually output binary directly, and these assembler listings can make very instructive reading.

2.6 Loading and linking

A typical compiler will not actually produce final machine instructions in specific places in the memory of the target machine. Compilers usually produce a list of machine instructions, in the order in which they are to be executed, but

with **relocatable addresses**. Some of the binary words are distinguished and the value of the starting address of the code must eventually be added.

As a compiler will normally produce **relocatable code**, there has to be an associated program called a **loader** whose job it is to actually place the target code in to memory, and alter the values of the relocatable addresses. It also has to place the data in the proper locations in memory. The loader produces the final **executable** target code.

The loader usually also performs **link-editing** – the adding together of several compiled programs to form one program in which all the addresses are properly synchronized. This may well involve calls to a library and the selection of the particular routines from that library that are required by the program.

2.7 Passes and relations between the phases

There are many ways in which a compiler can read a source program, and these will depend to some extent on the nature of the source language. For example, if the language allows the use of identifiers before their values have been given then the compiler may actually scan the source code twice, the first time constructing a table which contains the values of the identifiers when they eventually appear and then a second time to actually interpret the code. This is often referred to as a 'two-pass' compiler. It is possible, of course, to do the compiling in one pass by having some way of temporarily translating undefined identifiers, or leaving 'holes' in which they will later be inserted, and going back and substituting the actual value once it is discovered. This is sometimes referred to as **backpatching**.

Compiling is often done in several **passes**, during each pass the compiler reads the input file and produces an output file. This can be done in many different ways. It is possible, for example, to have the lexical, syntactic, and semantic analysis all done in one pass with the syntax analyser acting as 'manager'. In this case, when the syntax analyser requires the next token it calls up the lexical analyser to find it. When it has recognized an expression the syntax analyser then calls a routine to perform the semantic analysis and produce the intermediate code. Thus on a single pass a program in the intermediate code has been produced. The next pass then takes this as input.

Clearly there is also a balance to be met when deciding how many passes are desirable. On the one hand passes are time consuming because they require reading and writing of input and output files, but on the other hand reducing the number of passes usually increases the amount of the program that must be held in memory at any one time.

2.8 Input preprocessing – buffers

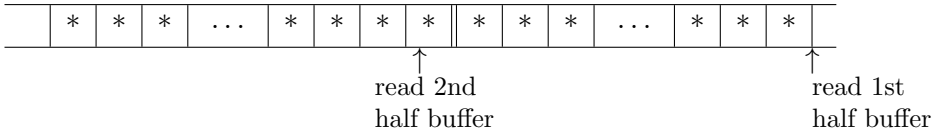
The need for careful use of buffers for inputting data to the lexical analyser arises because of the need for **lookahead**. It may be necessary to read several characters of a keyword, or even beyond the keyword, before it can be recognized.

For example, in C++ most keywords can be distinguished by looking ahead at just one or two characters. The + sign can be recognized as soon as it is read, whilst if the first character read is < it is only necessary to look ahead one more character to decide whether this is the less-than symbol or, if it is followed by say =, that it is the less-than-or-equal symbol. For reading numbers we need to look further ahead since we don't know until we reach the decimal point that 1234543.78 is a float rather than an integer.

The most obvious input method is to have the lexical analyser read in characters one at a time until the input string matches a keyword. However, we may need to look ahead beyond the keyword before it can be recognized, so extra characters have to be read and then pushed back into the input stream once a keyword is recognized. These characters then have to be read again when it is their turn to be identified. The problem with this is that reading characters in from input is very time consuming, and the same character may end up being read several times during one input. We use teo pointers, one of which points to the beginning of the current word and the other moves forward until the whole word is read.

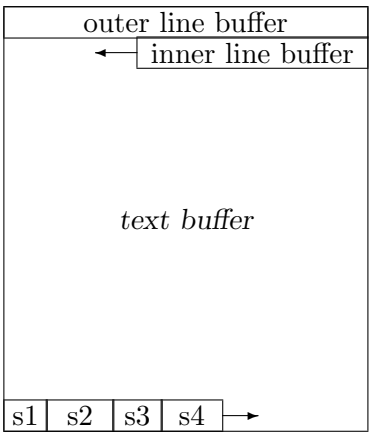
The problem then is: what happens when the forward pointer moves passed the end of the buffer? A common technique to deal with this is a **two-buffer** input scheme. In such a scheme the buffer is thought of as being divided into two equally sized halves, and there are two pointers to the buffer. Initially both pointers point at the beginning of the buffer. One, the forward pointer, moves forward character by character along the buffer until keyword matches the string between the two buffers. Once this match is made the forward pointer is placed at the end of the string and between the two pointers we have a keyword, which is now processed by the lexical analyser. Once this is done both pointers are moved to the first character beyond the keyword and the process begins again.

When the forward pointer is about to move past the half-way point in the buffer then the next block of input is read into the second half of the buffer and the identifying process continues. When the forward pointer is at the end of the buffer then the next block of input data is read into the first half of the buffer and the forward pointer is 'wrapped' round to point at the beginning of the buffer again.



2.9 More on input buffering

Text buffering is a surprisingly troubling part of lexical analyser design. Supporting nested include files, source echoing and synchronised error messages requires careful design. The text buffer manager maintains a single large area of memory. New strings can be inserted at low addresses and grow upwards.



The top of the region is used as a pushdown stack of line buffers for the set of included files. As each nested include file is opened, a record containing the previous state of the text manager is pushed onto a linked list and a new line buffer opened up. At the end of the included file, the buffer is released, the record list popped and scanning continues where it left off. End of file is not returned to the caller until the outermost file is consumed.

This arrangement allows arbitrary strings of arbitrary lengths to be stored, and files with arbitrarily long lines to be read. As each new line is read in, it is stored backwards at the top of the buffer. The lexer does not run out of memory until the strings meet the line buffers, so memory can always be fully used.

3 Lexical Analysis

Just as in natural languages, computer languages are composed of words which can be combined in certain ways to form sentences (program fragments). From a compiler's point of view, these sentences are strings and the strings are specified by a formal grammar. The grammar has terminals and non-terminals, and the strings are sequences of terminals that can be derived from the start symbol.

The terminals of the grammar are essentially the keywords, identifiers and literals of the programming language. The program that the compiler is to translate is composed of characters which can be entered from a keyboard – letters, digits, spaces, carriage returns, punctuation symbols etc. These symbols need to be converted into keywords/terminals before the compiler can check whether the input program is syntactically correct.

Grouping the input characters into terminals is usually done by a *lexical analyser*. At the lexical analysis stage, the terminals are usually called *tokens*. It is not sensible to have a separate token for all of the words; we don't want each identifier and each number to correspond to a different token. So a token can correspond to several words. The words are strings of characters, the set of strings which corresponds to a particular token is called the *pattern* of the token, and a string from the pattern is called a *lexeme* of the token.

The lexical analyser reads in characters from the input file until it recognises the lexeme of a token. It then stores the lexeme in the symbol table and returns the token corresponding to the lexeme. (Although in certain cases, such as when the lexeme is a keyword such as 'if' or 'while', there may be no need to store the lexeme in the symbol table.) Comments and layout information are removed from the program by the lexical analyser.

In this section we shall discuss the use of regular expressions to specify the patterns of tokens, and the use of finite state automata to recognise the lexemes. We give a formal method for constructing an NFA corresponding to a particular regular expression, a procedure (the subset construction) for converting this NFA into a DFA, and a procedure for improving the efficiency of this DFA.

3.1 Tokens

One of the first steps in writing a lexical analyser is to decide exactly what the tokens it recognizes are to be. This includes deciding what types of tokens there are and what the name of each token is.

In the final implemented compiler the tokens will be represented by integers, but we will give these integers mnemonic names, e.g. `ID`, `IFSYM`, `ELSESYM`, `NUM`, `DIGIT`, etc. (These are the objects that are printed in bold type in Aho et.al.) Associated with each token is a set of strings (lexemes). When a string which is a lexeme is input to the lexer a corresponding token is output.

The set associated with the token `IFSYM` has only one element (the reserved word 'if'), so the associated set is { if }. But for some tokens the set may be very large, the set corresponding to `NUM` may be $\{\ldots -2, -1, 0, 1, 2, 3, \ldots\}$. For some tokens it is not easy to describe the associated set in formal language,

e.g. the set associated with `ID` may be the set of all finite strings starting with a letter and followed by digits and letters, excluding strings like if that are reserved words! How would you describe this set formally? (We shall discuss this in the next section.)

The compiler must keep track of the actual string that was input from the source program, not just the token to which it corresponds. If the identifier 'sum' is read we want to know more than just that it is an `ID`, we want to know which one. In practice several pieces of information may be stored, and these are called the **attributes** of the token. For example, the lexeme of the token will be kept if it cannot be deduced from the token alone, the line number on which the token was first met is also often kept for error reporting purposes, and the token may have, or later be assigned, a value in which case this is also stored.

3.2 Regular Expressions

An **alphabet** is any set of symbols. A string in the alphabet A is any sequence of elements of A. For two strings α and β in A we write $\alpha\beta$ for the concatenation of α and β, the string obtained by writing the elements of β after the elements of α. We let ε denote the empty string and A^* denote the set of all strings, including ε in A; A^* is the Kleene closure of A.

For all α in A^* we have $\alpha\varepsilon = \alpha = \varepsilon\alpha$. Of course, we will use the usual abbreviations like a^3 for aaa etc.

For sets A and B we write

$$AB = \{ab \mid a \in A, b \in B\}, \quad A^+ = A^* \backslash \{\varepsilon\} \quad \text{and} \quad A^3 = \{a_1a_2a_3 \mid a_i \in A\}.$$

So AB is the set of all strings of length 2 which have one element of A followed by one element of B, A^+ is the set of all non-empty strings in A, and A^3 is the set of all strings in A of length 3. We also get that $B(A \cup (B^2))^*$ is the set of all strings that begin with an element of B and is then followed by a (possibly empty) string of elements of A and pairs of elements of B.

Given an alphabet, A, we then define **regular expressions** over A.

1. The element ε is a regular expression which denotes the set containing just the empty string ε.

2. For $a \in A$, a is a regular expression which denotes the one element set $\{a\}$.

   If r and s are regular expressions then:

3. rs is a regular expression which denotes the set of strings which are formed by concatenating a string from r with a string from s.

4. r^* is regular expression which denotes the set of strings which are sequences of zero or more strings from r concatenated together.

5. $r|s$ is a regular expression which denotes the union of the sets r and s, the set of strings which are either a string from r or a string from s.

Thus $a|b = \{a, b\}$, $(ab)^* = \{\varepsilon, ab, abab, ababab, \ldots\}$, $ab^* = \{a, ab, abb, \ldots\}$ and $(ba|c)^*d$ is the set of strings end with d and have some c's and ba's in front, eg $cbaccbad \in (ba|c)^*d$ but $babcd, bac \notin (ba|c)^*d$. The | operator has lowest priority and $*$ has highest, so $r \mid t\ s^* = r \mid (t\ (s^*)\ )$.

We should also recall that there may be many ways of denoting the same regular expression. For example, $a|a$ and a both denote $\{a\}$ and $a(b|c)$ and $ab|ac$ both denote $\{ab, ac\}$.

We call the set of strings denoted by the regular expression r the **language** denoted by r. Two regular expressions, r and s, are said to be **equivalent** if they denote the same language. In this case we write $r = s$.

$r|s = s|r$, $r|(s|t) = (r|s)|t$, $r(s|t) = rs|rt$, $r|\varepsilon = r$, etc.

3.3 Tokens and regular expressions

For a computing language the alphabet usually contains digits 0, 1, ... , 9; letters A, B, ... , Z, a, ..., z; parentheses (,); and other symbols =, +, -, %, $, @, !, etc.

It is (essentially) possible to define the language of each token using a context free grammar, and to use the same techniques for lexical analysis as we shall use later for syntax analysis. For various reasons which include efficiency, ease of error reporting, the hiding of non-LL(1) or non-LALR(1) details and free format normalisation, it is most common to use regular expressions to define tokens.

For notational purposes it is useful to extend our constructs to include these shorthands:

1. r^+ (positive closure, or one-or-many occurrences) is a regular expression denoting $r\ r^*$.
2. $r?$ (zero-or-one occurrences) is a regular expression denoting $r \cup \{\epsilon\}$.
3. $[abc]$ (character class) is a regular expression denoting $\{a, b, c\}$.
4. $[a - d]$ (character class range) is a regular expression denoting $\{a, b, c, d\}$.
5. $[\neg a - c]$ (character class range) is a regular expression denoting $L \backslash \{a, b, c\}$.

We often give names to regular expressions. For example,

$$0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

is a regular expression that denotes the language $\{0, 1, \ldots, 9\}$. We often call this expression *digit*. We write the rule

$$digit \quad \rightarrow \quad 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9.$$

This is read as saying 'digit is the name of the regular expression $0 \mid 1 \mid \ldots \mid 9$'.

A **regular definition** over a language L is a set of rules

$$\begin{array}{lcl} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \end{array}$$

$$\begin{array}{lcl} d_3 & \rightarrow & r_3 \\ & \vdots & \\ d_n & \rightarrow & r_n \end{array}$$

where the d_i are (distinct) names for regular expressions over L and the r_i are regular expressions over $L \cup \{d_{i+1}, \ldots, d_n\}$.

For example, if L is the set of all digits and upper and lower case letters, so

$$L = \{0, 1, \ldots, 9, A, B, \ldots, Z, a, b, \ldots, z\},$$

then `ID`, the set of all identifiers in C, can be described as a regular expression over L. We have the following regular definition for `ID`:

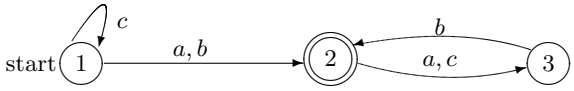
$$\begin{array}{lcl} id & \rightarrow & letter(letter \mid digit)^* \\ letter & \rightarrow & A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z \\ digit & \rightarrow & 0 \mid 1 \mid \ldots \mid 9 \end{array}$$

3.4 Finite state automata and lexical analysis

It is possible, and instructive, to give graphical interpretations of regular expressions.

A **finite state automaton** (NFA) (also called a **transition diagram**) is a directed graph. The nodes of the graph are called **states** and the nodes are connected by arrows (transitions) that are labelled with letters from an alphabet. One state is identified as the **start state**, and certain others are nominated as **accepting states**. A string is said to be identified by an NFA if by starting at the start state and moving along arrows labelled by the characters in the string we end up at an accepting state.

For example, in the following NFA the accepting states are denoted by double circles, and the NFA accepts the string *caabcb* but not the strings *caabc* or *cab*:



This method can do more than just recognize elements. It can report that the string *caabc* was not of the required form, and it can also say that it was expecting a *b* to be input, because this is the only input that causes transition out of state 3.

We say that r is **the regular expression represented by the NFA** if the strings accepted by the diagram are exactly the strings in the language denoted by r.

The above NFA represents the expression $c^*(a|b)((a|c)b)^*$.

The following NFA accepts exactly those strings in the language of the regular expression `ID` defined in the previous section.

We can express the action of an NFA using a **transition table**. This is a table that has a row for each state of the NFA and a column for each possible input symbol. The entries in the table are the states that can be reached from the given state by inputting the specified symbol. (If there is no such state then then symbol – is entered.) Note, sometimes, for example during dead state minisation, transition tables are written the other way round, i.e. transposed, shown on the right below.

The transition table for the first NFA above is:

	a	b	c
1	2	2	1
2	3	–	3
3	–	2	–

	1	2	3
a	2	3	–
b	2	–	2
c	1	3	–

A **deterministic finite-state automaton** or **DFA** is a finite-state automaton for which, for each state and input symbol there is at most one state into which the machine can go.

We allow arrows to be labelled by the empty string, this means that the machine can move from one state to the next via an empty arrow without requiring any input. A DFA cannot have transitions labelled ϵ.

When an NFA gets input that results in two or more possible choices of action effectively the machine replicates itself and produces one machine for each possible action. Each of these machines run in parallel, reproducing themselves further if more choices are reached. We describe this as the original machine going into several states in parallel. The machine accepts the input string if any of the parallel replications ends up in an accepting state. Thus, the entries in the translation table for an NFA are sets of states.

Two finite-state automata are equivalent if they correspond to the same regular expression.

Formally an NFA consists of a set, S, of states; a set, L, of input symbols together with the special symbol ε; a function, *move*, that maps a pair of the form (state, symbol) to a set of states; a specified start state s_0; a specified subset, N, of states called accepting states.

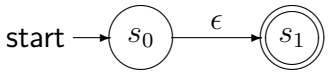
The lexical analysers that we are going to describe use DFAs as the mechanism for recognising token lexemes. A significant consequence of the theory underlying compiler generation is that it is possible to write a program which, given an appropriate specification of a language, will *automatically* generate a compiler for the language. Such a 'compiler-compiler' will include a lexical analyser generator which will have to generate a DFA from a given regular expression. Thus we need an automatic method of generating DFAs from regular expressions – one which can be programmed.

We shall now describe such a method which first constructs an NFA using Thompson's construction, then generates an equivalent DFA using the subset construction, and finally attempts to generate a more efficient DFA.

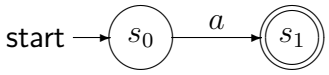
3.5 Thompson's construction

As we have already said, it is important to have a mechanical method of constructing an NFA corresponding to a given regular expression; a method which will always work and does not require any 'intuition'. We use the inductive definition of a regular expression to automatically build a corresponding NFA.

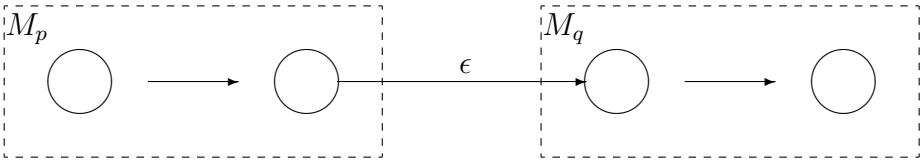
The regular expression ϵ corresponds to the NFA



The regular expression a, where $a \in L$, corresponds to the NFA

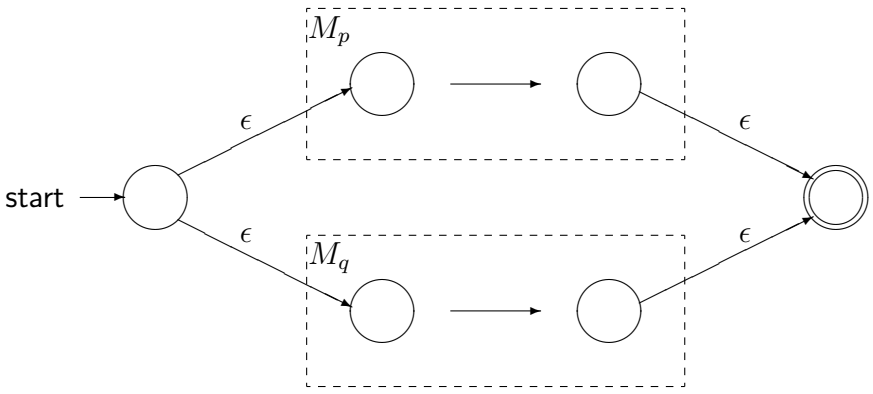


If p, q are regular expressions with corresponding NFAs M_p, M_q then:
pq (concatenation) is represented by



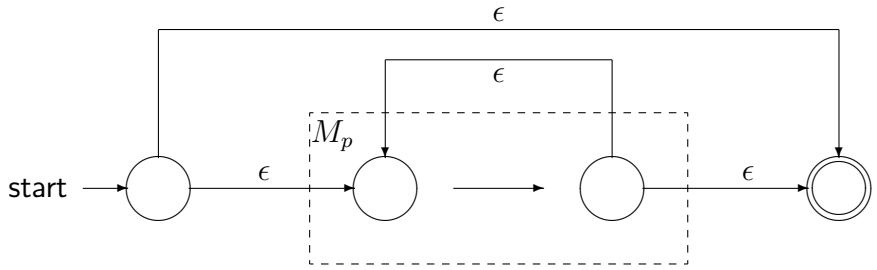
that is the two machines in series joined by an empty transition.

$p \mid q$ (alternation or union) is represented by



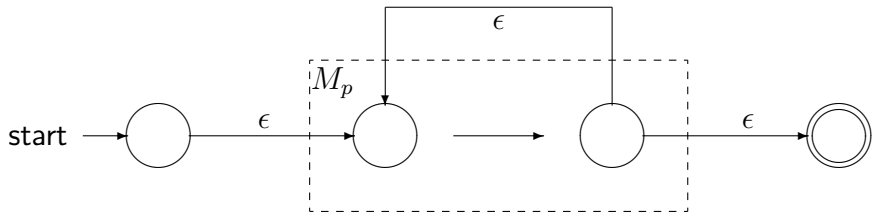
the two machines in parallel joined by empty transitions to a new start state and a new final state.

p^* (Kleene closure) is represented by

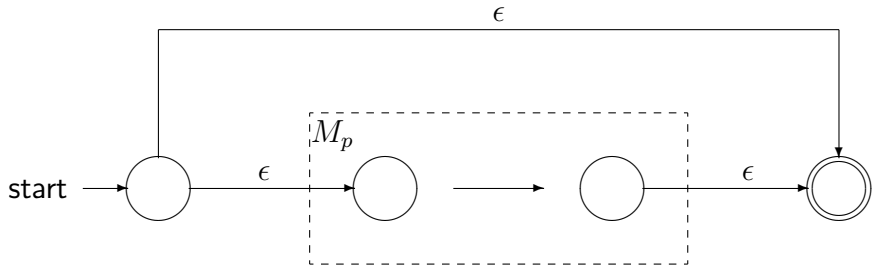


the machine for p with new start and finish states, all joined by empty arrows.

p^+ (positive closure) is represented by



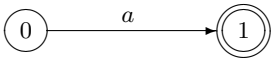
p? (optional) is represented by



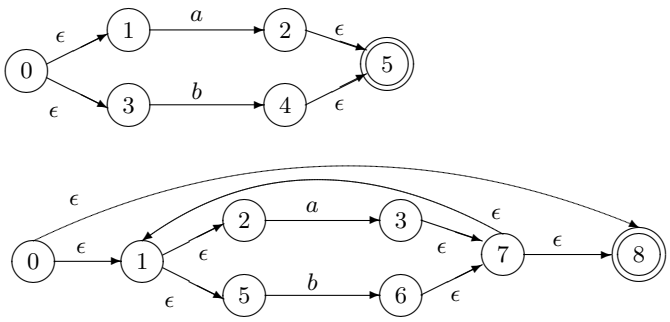
3.6 CS1870 material

The construction of a DFA from a regular expression was discussed in CS1870. In this section part of the CS1870 notes are repeated.

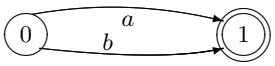
It is easy to see that the following DFA, whose start state is 0, has language $\{a\}$, the set denoted by the regular expression a.



It is also easy to see that the following FAs have languages $(a \mid b)$ and $(a \mid b)^*$ respectively.



These aren't the most obvious FAs. If we just wanted the FA for $(a \mid b)$ we would probably write

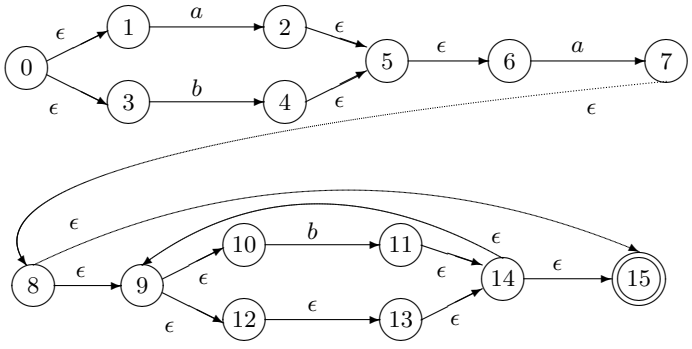


But the point is that the above FAs can be constructed by a formula that applies to any regular expression. (If r and s are regular expressions for which we already have FAs then we can generate an FA for the regular expression $(r \mid s)$ by making new start and accepting states and join these to the start and accepting states of the FAs for r and s, see below). If there is a formula for the construction then this can form the basis of a computer program which *automatically* constructs an FA for a regular expression.

Once the computer can construct the FA we can also write a program to traverse the FA with an input string and we will have a program the can recognise the words of a programming language!

The formula for constructing an FA for a regular expression is inductive and uses the inductive definition of regular expressions.

Example Using Thompson's construction for the regular expression $(a \mid b)\ a\ (b \mid \epsilon)^*$ we get the FA



3.7 The subset construction

The automata constructed above are *non-deterministic finite automata*. They can have several arcs with the same label leaving a node, and ϵ arcs are also allowed. In reality a sequential computer can not cope directly with this non-determinism.

To directly simulate the behaviour of an NFA we convert it to a *deterministic finite automaton* (DFA) which has no ϵ arcs and at most one arc with any given label leaving a node. It might seem intuitively that an NFA should be more 'powerful' than a DFA, but in fact there is a DFA for every NFA.

In this section we give a general algorithm which takes an NFA and produces an equivalent DFA (one which recognises the same set of strings).

Formally an NFA consists of a set, S, of states; a set, L, of input symbols together with the special symbol ϵ; a function, *move*, that maps a pair of the form (state, symbol) to a set of states; a specified start state s_0; a specified subset of states called accepting states.

The NFA is a DFA if for each state $s \in S$ and input symbol $a \in L$ the set $move(s, a)$ contains at most one element, and the set $move(s, \epsilon)$ is always empty.

Let N be an NFA, and let D denote the DFA we are trying to construct. The states of D are sets of states of N. The sets are generated by looking for the subset of states that can be reached by making any number (including zero) ϵ moves from states in N.

We need the following notation. If s is a state in our NFA, T is a set of such states, and $a \in L$ then

$\epsilon - closure(T)$ is the set of NFA states reachable from NFA state s in T *via* ϵ-transitions alone.

Ta is the set of NFA states to which there is a transition on input symbol a from some state s in T. So $Ta = move_N(T, a)$.

The algorithm for constructing the required DFA D is as follows:

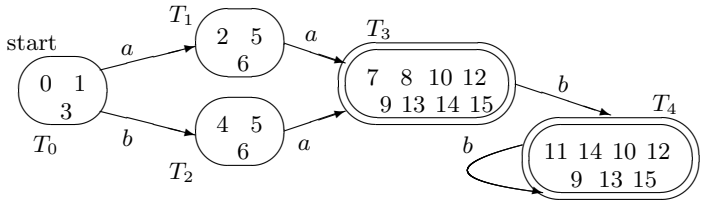
The start state, T_0, of D is ϵ-closure($\{s_0\}$), the set of states which can be reached from the start of N by just using ϵ arrows. Then for each $a \in L$, calculate T_0a – the set of states from N which can be reached from some state in T_0 along an arrow labelled a. If T_0a is not empty, form the ϵ-closure of this by adding all the other states which can be reached from states in T_0a along ϵ arrows. So we have a new state $T_1 = \epsilon$-closure(T_0a) in D. We then put an arrow labelled a from T_0 to T_1 by defining $move_D(T_0, a) = T_1$, and 'mark' the state T_0 as having been dealt with. We then repeat the process for each one the new states that we have constructed. So for every $a \in L$ we form T_1a, the set of states reachable using an a arrow from a state in T_1, and then we calculate ϵ-closure(T_1a) etc. When all the states T_i that we have constructed are also marked as having been dealt with then the construction of D is complete.

The start state of D is the state T_0 which contains the start state of N. A state in D is an accepting state if it contains at least one accepting state from N. We call the transition function for D $moveD()$ to distinguish it from the move function for N.

Example from CS1870

The subset construction on the NFA for $(a \mid b)\ a\ (b \mid \epsilon)^*$ given above gives the following sets.

$$
\begin{array}{rll}
T_0 = \epsilon\text{-closure}(\{0\}) = \{0,1,3\} & T_0a = \{2\} & T_0b = \{4\} \\
T_1 = \epsilon\text{-closure}(T_0a) = \{2,5,6\} & T_1a = \{7\} & T_1b = \emptyset \\
T_2 = \epsilon\text{-closure}(T_0b) = \{4,5,6\} & T_2a = \{7\} & T_2b = \emptyset \\
T_3 = \epsilon\text{-closure}(T_1a) = \{7,8,9,10,12,13,14,15\} & T_3a = \emptyset & T_3b = \{11\} \\
\quad \epsilon\text{-closure}(T_2a) = T_3 & & \\
T_4 = \epsilon\text{-closure}(T_3b) = \{11,14,15,9,10,12,13\} & T_4a = \emptyset & T_4b = \{11\} \\
\quad \epsilon\text{-closure}(T_4b) = T_4 & &
\end{array}
$$



Pseudocode algorithms

The following is a formal algorithm for the above process.

```
Dstates := epsilon-closure(s0)
while there is an unmarked state T in Dstates do
  mark T
  for each input symbol a do
    U := epsilon-closure(move(T,a))
    if NOT U IN Dstates then
      add U to Dstates
    moveD(T,a):=U
```

Computing an ϵ-closure requires a graph search algorithm. Such algorithms usually explore the graph by first pushing start nodes, and then popping a node, exploring its neighbours and pushing them if they meet the search criterion.

```
push all states in T
initialise epsilon-closure(T) to T
while NOT stack empty do
  pop t
  for each state u with an edge from
    t to u labeled epsilon do
       if NOT u IN epsilon-closure(T)
         push u
         add u to epsilon-closure(T)
```

3.8 Minimising a DFA

The procedure described so far may not give the most efficient DFA (i.e. the one with the fewest states), so we consider an algorithm to minimize DFA's.

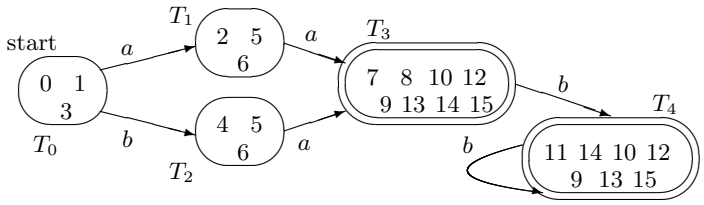
AIM: Given a DFA $(S, L, move, s_0, A)$ find an equivalent more efficient DFA $(S^{new}, L, move^{new}, s_0^{new}, A^{new})$

Suppose that we have a DFA with states S and accepting states $A \subseteq S$. We begin by adding a new 'dead' state d and adding arrows from every other state to d until there is an arrow labelled with each element of L. Thus we extend the function $move$ so that, for all $s \in S' = S \cup \{d\}$ and $a \in L$,

$$move'(s,a) = \begin{cases} move(s,a), & \text{if this exists,} \\ d, & \text{otherwise.} \end{cases}$$

Start by dividing S' into two disjoint partitions $S_1 = A$ and $S_) = (S \backslash A) \cup \{d\}$. We define the process inductively. Suppose that we have already partitioned S so that we have disjoint subsets $S_0, \ldots, S_n$. For each set S_i we subdivide as follows: two elements $s, t \in S_i$ stay in the same subpartition if and only if for all $a \in L$ for some j, $move'(s,a)$ and $move'(t,a)$ both lie in S_j. When the partition cannot be divided any further the process stops and we discard the state containing d. If $T_0, \ldots, T_m$ are the partitions constructed at the time the process stops we choose one state from each set and use these to construct the new DFA. Thus the new DFA has states $S^{new} = \{T_0, \ldots, T_m\}$; input symbols L as before; the function $move^{new}$ which is essentially the restriction of the old $move$ to the new set S^{new}, so $move^{new}(T_i, a) = T_j$, where, for $t_i \in T_i$, $move(t_i, a) \in T_j$; the new start state is T_i, where $s_0 \in T_i$, and T_i is an accepting state if it consists of accepting states from D.

Example



Start with

$$S_0 = \{d, T_0, T_1, T_2\} \qquad S_1 = \{T_3, T_4\}$$

Build a move table in which the entry states are replaced with the set S_0 or S_1 to which they belong.

	d	T_0	T_1	T_2	T_3	T_4
a	S_0	S_0	S_1	S_1	S_0	S_0
b	S_0	S_0	S_0	S_0	S_1	S_1

Split the states again

$$R_0 = \{d, T_0\} \qquad R_1 = \{T_1, T_2\} \qquad R_2 = \{T_3, T_4\}$$

and build the move table

	d	T_0	T_1	T_2	T_3	T_4
a	R_0	R_1	R_2	R_2	R_0	R_0
b	R_0	R_1	R_0	R_0	R_2	R_2

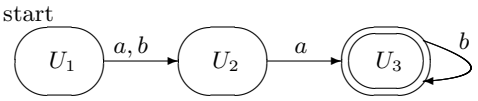
Split the states once more

$$U_0 = \{d\} \qquad U_1 = \{T_0\} \qquad U_2 = \{T_1, T_2\} \qquad U_3 = \{T_3, T_4\}$$

and build the move table

	d	T_0	T_1	T_2	T_3	T_4
a	U_0	U_2	U_3	U_3	U_0	U_0
b	U_0	U_2	U_0	U_0	U_3	U_3

This is now stable giving the minimum DFA



3.9 A lexical analysis algorithm using a DFA

It is simple to give an algorithm in pseudo code to recognize a string of input characters as being in the pattern for a regular expression described by a DFA which has start state s_0, function *move* and accepting states N. We assume that we have a routine *nextch()* that reads and returns the next symbol from the input string

```
s := s0
c := nextch()
while  move(s, c) is non-empty  do
       s := move(s, c)
       c := nextch()
       end
if  s lies in N  then  return  "YES"
else  return  "NO"
```

3.10 A lexer for the whole language

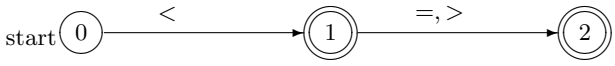
We now have what we need to build a lexical analyser provided that we can define the patterns of the tokens of the source language by a regular grammar.

To make a pattern recogniser for the whole of the source language we write separate state machines for each token and then join the algorithms together so that the machine works through them until it finds the appropriate one. We need to think about the order in which we put the algorithms to make sure that we get the right behaviour.

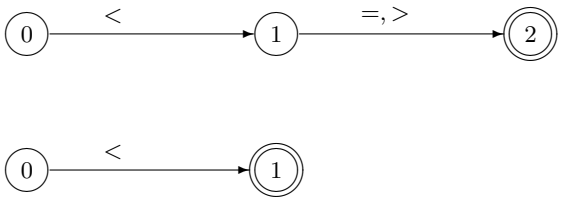
Suppose that rather than having separate tokens for each of the symbols $<$, $<=, <>$ we have one token, say `REL`, so that a regular definition for `REL` is

$$rel \quad \rightarrow \quad < \ (\varepsilon \mid = \mid >).$$

Then a DFA that recognizes the pattern of `REL` is:



Using this alone we cannot get our lexical analyser to do the 'lookahead' that is needed. A solution is to actually have two DFA's to recognize the pattern for `rel`, and to put them in order.



When < is read as an input character the lexical analyser first tries the top DFA. If it gets a match then the token `REL` is returned together with the lexeme of current input characters. If the routine returns a non-match then the current characters are pushed back onto the input stack and the lexical analyser begins again this time using the second DFA.

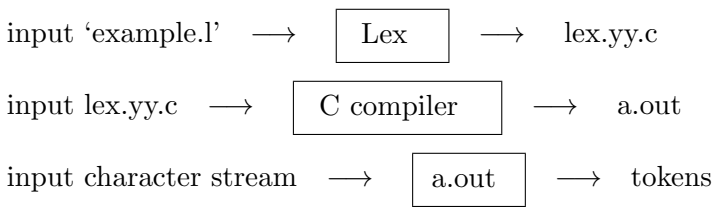
This also allows us to distinguish between reserved words and identifiers simply by getting the routine to check for reserved words first.

3.11 Lex

Lex is a lexical analyser generator – a software tool that takes as input a specification of a set of regular expressions together with actions to be taken when an expression is recognized. The output of Lex is a program that recognizes the regular expressions and takes the appropriate action. When the input expressions define the tokens of a programming language the output can be used as a lexical analyser of a compiler for the language. Lex works by transforming the regular expressions into equivalent DFA's. It also provides a routine to read the input characters.

The actions to be taken are written in C.

Using the editor, create a file 'example.l' that is the input to Lex. Then type 'lex example.l' and Lex creates the file lex.yy.c which is written in C. This file is then run through a C compiler, you can use the command 'gcc lex.yy.c -lfl', which produces a program called a.out. This is the lexical analyser.



An input file for Lex, a Lex program, has three parts: declarations, rules and auxiliary procedures. Each part of the program is separated by a line '%%'.

Lex program format:

declarations
%%
rules
%%
auxiliary procedures

Any of the three parts may be empty but the first line of separators cannot be left out.

The declarations section includes declarations of variables and any regular definitions that have been used as components of the regular expressions in the rules section. We can also declare an identifier to represent a constant.

The rules section is a list of statements of the form p $\{action\}$, each statement must be on a new line. *action* is a program fragment written in C that describes what should be done if a lexeme input matches the pattern p.

The third section contains any auxiliary procedures that may be needed by the actions.

The following is a very simple Lex program that generates a lexical analyser which recognizes the symbols <, <=, <> as being lexemes for the token `REL`. There are no auxiliary procedures and no regular definitions.

```
%{
#define rel 6
%}
rel <(=|>)?
%%
{rel}   {printf("rel") ;}
%%
```

Lex recognizes various additional regular expressions that can be thought of as abbreviations for formal regular expressions. In particular r? is $(\varepsilon|\mathrm{r})$.

Create a file ex1.l, then type lex ex1.l. This produces lex.yy.c, so type gcc lex.yy.c -lfl to produce a.out. Then if you type a.out and then < you will get 'rel' printed on the screen. Unrecognized character strings are echoed. So if you type 'and' then 'and' will be repeated back to you.

Anything in the lex program appearing between a pair of brackets of the form %{ and %} is copied directly into the output lex.yy.c. When constants are declared the declarations are surrounded by such brackets.

The Lex language has many metasymbols. All the normal symbols needed for regular expressions have their expected meaning. The expression [A–Z] is understood by Lex to mean the set of all capital letters. A space is interpreted as a blank space and \t and \n represent a tab and a newline respectively. If we want a character which is a metasymbol to have its natural meaning then we precede it by \. So to write a minus sign we need to write \–. Another way to give characters their natural meaning is to enclose them by quotation marks. If a string of characters is a name for a regular expression rather than the expression itself then the string must be enclosed in braces. Lex then looks back to the declared regular definitions to find the actual expression.

The following is a list of 'Lex regular expressions'. These should be thought of as abbreviations for formal regular expressions. These are the expressions that can be used in the regular definitions that define the patterns in the rules section of the Lex input file.

```
c                any non-operator character
\c               the character c literally
's'              the string s literally
.                any character except newline
^                beginning-of-line
$                end-of-line
[s]              any character in the string s
[^s]             any character not in the string s
r{n,m}           n to m occurances of r
(r)              r
r/s              r when followed by s
[a-z]           the set of all lower case letters
[A-Z]           the set of all upper case letters
[0-9]           the set of all digits
\t               tab
\n               newline
```

If two substrings match a pattern, the longer is chosen. So if you type <= <= it will return rel rel.

Lex is designed to be run as a subroutine for a parser generator such as Yacc. In this case the action on recognizing a token is 'return *token*' which returns the token to the parser that has called Lex. Thus the only output of the lexical analyser is the token.

To pass an attribute value such as the lexeme we can set a global variable, called yylval by Yacc, in which is placed the constant corresponding to the lexeme.

```
%{
#define plus 3
#define times 4
#define op 5
#define rel 6
%}
rel <(=|>)?
%%
'+'   {yylval = plus; return(op ) ;}
'*'   {yylval = times; return(op ) ;}
{rel} {yylval = install_rel(); return(rel ) ;}
%%
install_rel()  {/* code for this procedure */}
```

There are two ways to get the lexical analyser to write lexemes to yylval. One is to declare the name of a lexeme as a constant, then a rule is declared for

each lexeme that says when the lexeme is input the value of the constant is written to yylval and the appropriate token is returned. The other is to provide a procedure, written in C, that returns the value of the appropriate lexeme. This is listed in the auxiliary procedures section of the input. The code is put between braces and will be copied verbatum into lex.yy.c.

Example: This is part of the input to generate a lexical analyser for a language where identifiers can be any non-empty string of letters except for the keywords 'if', 'then' and 'else'. It ignores any sequence of spaces, tabs, and newlines. The statements in each action are separated and terminated by semi-colons.

```
%{
#define plus 3
#define times 4
#define op 5
#define rel 6
#define ifsym 7
#define thensym 8
#define elsesym 9
#define id 10
%}
rel           <(=|>)?
delim         [\t \n]
ws            {delim}+
letter        [A--Za--z]
id            {letter}{letter}*
%%
{ws}          { ;}
if            {return(ifsym) ;}
then          {return(themsym) ;}
else          {return(elsesym) ;}
{id}          {yylval = install_id(); return(id) ;}
'+'           {yylval = plus; return(op) ;}
'*'           {yylval = times; return(op) ;}
{rel}         {yylval = install_rel(); return(rel) ;}
%%
install_id() { /* code for this procedure */}
install_rel() { /* code for this procedure */}
```

Lex has various defaults in line with the theory that we have discussed in earlier sections. If the input to the lexical analyser created by Lex is matched by more than one of the expressions used to create the program then it automatically chooses the expression which matches the longest possible string of input. If there is still ambiguity then the expression given first is chosen to be the match. This is why the lexemes if, then and else cannot be identifiers.

There are many other features of Lex, we have only included a few here to give an idea of the package. If you want to know more you can read the relevant parts of Aho et.al. and the references given there.

One useful feature that we haven't discussed is the definition of tokens with contexts. The context does the job of a lookahead operator. The lexeme DO in FORTRAN represents the operator rather than an identifier (i.e. is dosym) if it is followed by an integer, an identifier, =, an integer and a comma. In the Lex language / denotes the context operator and we can write

DO / $\{digit\}(\{letter\}|\{digit\})^* = (\{digit\}^+|,)$

which recognizes the string DO if it is followed by a string in the regular expression following /.

3.12 Symbol tables

A *symbol table* is used to hold attribute information for identifiers. Attributes are used to resolve context sensitivities as well as for housekeeping. Useful attributes include

- ◇ the *type*,
- ◇ the *reference level* (constant, variable, pointer, ...),
- ◇ the line and column at which the identifier is declared,
- ◇ a list of lines and columns at which the identifier is used,
- ◇ a value (especially for constants, but also for other reference levels in an interpreter or optimising compiler),
- ◇ the scope.

We need functions to insert, delete and search for identifiers in a symbol table.

Here are declarations in C that might describe symbol table entries:

```
struct coordinate_struct {
  unsigned line;
  unsigned column;
  struct coordinate_struct* next;
};
struct sym_tab_struct {
  char * id;
  int scope;
  struct sym_tab_struct* type;
  unsigned reference_level;
  struct coordinate_struct instances;
  union {
    char c;
    float f;
    int i;
    void *p
    char *s;
  } data;
};
```

The most important field in the symbol table data type is the `id` which acts as a search key. In a language with multiple scope regions (and very few languages do *not* have more than one scope region) some way of representing the identifier's scope is needed. Here, we use an integer which is incremented as each new scope region is entered, giving each scope a unique integer identification.

Identifiers have a *type*, and indeed some identifiers are labels for user defined types! Hence, our type field is a pointer to another symbol table record that is the type record itself.

The `reference_level` might in some implementations be incorporated into the type record, but here we have an integer that records the number of indirections that are required to convert an identifier into a value, with a constant being level 0.

`instances` is used to keep track of the points at which identifiers are declared and used. This information is useful when error messages are being generated, and may also be used to construct a cross reference table of identifiers.

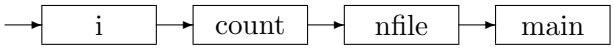
`struct coordinate_struct` holds a line number, a column number and a pointer to another coordinate. The field `instances` within `symb_tab_struct` holds the coordinates of the identifier declaration and the `next` field points to a linked list of coordinates at which the identifier is used.

The `data` union holds a variety of variables which may be used to hold values for constants, intermediate results of expression evaluation and so on.

Symbol tables need to be of arbitrary size, since we do not know in advance how many identifiers we will encounter during a particular compilation. We could allocate a very large array, and then issue a fatal error message when it fills up, but it is of course more space-efficient to use some kind of dynamic data structure to hold the symbol table.

Some compilers never delete an entry in the symbol table because the records are needed during the optimisation phases. Similarly, symbol entry is relatively rare compared to lookup because we would expect every variable to be used at least twice after declaration: at least once to write a value into it and at least once to read the value. In practice variables are used far more often. Empirical studies on Pascal programs indicate that symbol table searches are around 10 times more common than symbol table insertions.

Let us organise the symbol table as a single linked list, with new identifiers being added at the head of the list.

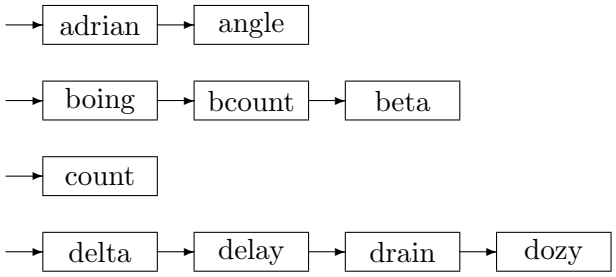


Here, insertion is very cheap, but searching is expensive. When the list has n elements we might expect an identifier to require $n/2$ comparisons before it is found.

Brinch Hansen described such a table built into a Pascal compiler. When working on a program with a 2,000 line program with 431 identifiers, the symbol table had to do on average 184 comparisons *per* identifier search. Symbol table lookup accounted for 25% of the execution time in this compiler. Of course,

longer programs would display even worse behaviour. Linear searching is thus unacceptable.

We could improve the performance of our linear list table by making one list for each letter of the alphabet, and inserting all the identifiers that begin with a particular letter on a separate list.



This might improve search times by a factor of 26, whilst keeping insertion time small. The catch is that identifiers are not evenly divided by initial letter.

If the identifiers could be distributed *randomly* (but predictably) between the n lists then we would indeed achieve a factor n speed up. This is the principle of the *hash table*.

3.13 Hash tables

A hash function is simply a calculation on a key that yields a random number. Hash functions are deterministic – that is they always yield the same number for a given key.

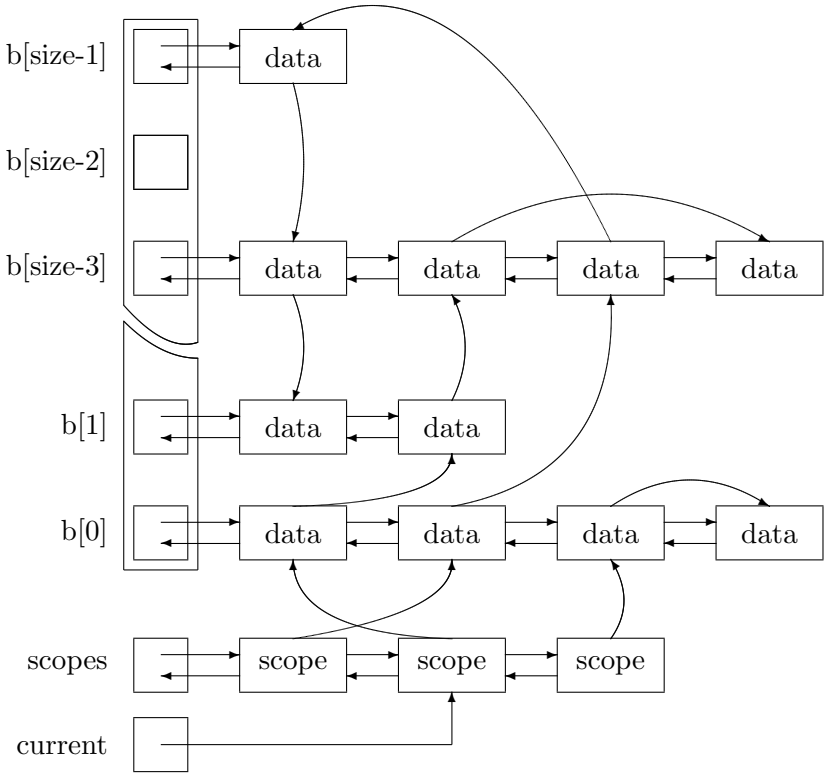
Perhaps the simplest hash function for a string is to add together the ASCII values for all of the characters in the string, and then take the modulus of the result with the number of sub-lists available. It turns out that this function works best if there are a prime number of sublists. An even better result is achieved if another number, coprime with the number of lists is factored in at each addition. This generates repeated overflows:

```
hashnumber := 0;
for i:=0 to length_of_string(atr) do
  hashnumber :== ord(str[i]) +
                 hashprime*hashnumber;
hashnumber := hashnumber MOD hashsize;
```

Most programming languages allow multiple scope regions. This, of course, means that there can be multiple identifiers with the same name but different attributes (otherwise there would be no point in having the multiple scopes).

The most common form of scope control is *nested* scope. Our organisation automatically supports nested scope as long as we start searching from the start of each list, since we will encounter the most recently declared instance of an identifier key first.

We need some way of marking scope regions, if only to allow for the removal of a scope region from the symbol table at the end of a scope region.



Each symbol table is described by a header record that contains pointers to a hash table, a scope list, various maintenance functions and some bookkeeping data.

Whenever a symbol is to be inserted into the table, its key fields are *hashed* generating a random number in the range 0 ... `size`. This hash number is then used to index into the hash table, selecting one of the linked lists. The symbol is then added to the head of the list. A lookup is performed by hashing the test symbol and then searching down the list for a match. Since the most recent additions are always examined first, the structure directly implements nested scope rules in that a new symbol will hide any symbols with the same key deeper in the table.

The hash lists are doubly linked so that symbols can be quickly unlinked from the chain.

Whenever a symbol is added to a hash list, it is also added to the head of the current scope chain. New scope may be declared, in which case a new scope record is created and added to the head of the scope list. The scope pointers are represented by curved arrows. Although not shown on the diagram, each symbol maintains a back link to its scope record allowing efficient checking of the scope level for a particular symbol. The current scope may be reset to a previously declared scope.

4 Syntax analysis I – top down parsers

Having converted the input stream into a sequence of tokens, the compiler now has to check that the token sequence does form a legitimate program, i.e. that the input is *syntactically correct*. In this section we shall look at methods for deciding whether an input program is syntactically correct. However, we should begin by noting that syntax analysis usually involves more than just this correctness check. If the input is correct then the syntax analyser may produce a 'derivation tree' which is the first step in the semantic analysis of the input. This reflects the fact that the syntax of a programming language is *chosen* to allow constructs with particular meanings to be built into the language. Thus in a real sense the syntax of a language is not separate from the semantics. The syntax is the first step in defining the semantics, which makes it natural for the output of the syntax analysis stage to be a structure from which much of the semantics of the input can be constructed.

In order for the syntax analysis of a programming language to be effective there must be a formal definition of the language. In this course these definitions are given using grammars. The syntax analyser (or *parser*) receives a sequence of tokens from the scanner and attempts to group these tokens into strings in the language. The tokens are the terminals of the grammar. There are many techniques which are used to recognise the syntactic structure of an input program, but none of the ones commonly used in practice work on all possible grammars, all possible context-free grammars, or even all grammars for actual existing programming languages. In fact grammars are classified according to various properties and then a parsing technique is used which is appropriate for a grammar of that type.

In the main part of this section and in Section 5 we shall describe the top-down recursive descent and bottom-up table driven parsing techniques, and we shall discuss the properties that grammars must have if these techniques are to generate correct parsers. Before doing this, we shall give a brief overview of parsing techniques in general and then we will review grammars and derivations and define derivation (parse) trees.

4.1 Parsing techniques and efficiency

When we discussed lexical analysis we gave a general technique which could be used on any set of tokens whose patterns could be described by regular expressions. The situation with syntax analysis is slightly different. There do exist general parsing techniques which will work on all context free grammars, but until recently these have not been practical. The best known *general* parsing algorithm displays a time complexity of $O(n^{2.37})$, although other algorithms are known that can execute in $O(n^3)$ with smaller constants of proportionality. Although these algorithms are theoretically interesting, traditionally they have been considered unsuitable for real programming language translators because even for short texts of around 1,000 elements running on fast computers which might take 1μs to perform each parser step, parse times will be in the region of a quarter of an hour. Simply doubling the size of the text to 2,000 elements raises

the time required to around two hours. To guarantee linear time complexity algorithms, we restrict our languages to those whose features allow them to be parsed in linear time.

It may be that this situation changes in the future because is has not been shown that general linear-time context free parsing is impossible, and also because nonlinear general parsing algorithms are becoming practical. It has been proved that there do not exist linear time algorithms for certain computer science related problems. In other areas, especially the theory of NP complete algorithms, there exist strong suspicions that such algorithms can not exist. In the field of context-free parsing algorithms, however, not only has nobody yet demonstrated that there can not be a general linear-time algorithm, nobody has produced a context-free grammar that could not parsed using an *ad hoc* linear-time algorithm. What is lacking is a general method for finding such parsers.

As we are concerned with automatic parser generation we shall approach the topic by studying two parsing techniques which are efficient and then studying the class of grammars to which each technique can be applied.

4.2 Grammars and languages

A *context-free grammar* is a 4-tuple $(N, T, S, \mathcal{P})$, where N is a finite set of non-terminal symbols, T is a finite set of terminal symbols, S is the start symbol and lies in N, and $\mathcal{P}$ is a set of production rules of the form $X ::= \beta$ where $X \in N \backslash T$ and β is a string of elements from N and T.
Example:

```
    terminals: 'if', 'else', 'stop', '=', '(', ')',
               'id', 'digit', 'skip'
non-terminals: stmt, exp
        start: stmt
        rules: stmt ::= 'stop'.
               stmt ::= 'skip'.
               stmt ::= 'if' '(' exp ')' stmt 'else' stmt.
               stmt ::= 'if' '(' exp ')' stmt.
                exp ::= 'id' '=' 'digit'.
                exp ::= 'id' '<' 'digit'.
```

When studying a programming language the terminals are the tokens of the language, and the terms *token* and *terminal* are synonymous.

The symbol | is reserved and is used to group together several productions with the same left hand sides. We shall describe grammars by listing their production rules, the production rule for the start symbol being listed first.

E::=D | E∗D | E+D
D::= 0|1|2|3|4|5|6|7|8|9

is the language with terminals + ∗ 1 2 3 4 5 6 7 8 9, non-terminals E and D, and start symbol E.

This formal metalanguage for describing a language is called Backus-Naur Form (BNF). It was introduced and used in the definition of Algol 60.

Let G be any grammar, let x be non-terminal and α and β be strings of non-terminals and tokens. We write

$$\alpha x \beta \Rightarrow \alpha \gamma \beta$$

if $x ::= \gamma$ is a production rule in G. We write

$$\delta \stackrel{*}{\Rightarrow} \gamma$$

if there exist α_i such that

$$\delta \Rightarrow \alpha_1, \quad \alpha_1 \Rightarrow \alpha_2, \quad \ldots, \quad \alpha_n \Rightarrow \gamma.$$

We allow $n = 0$ so $\delta \stackrel{*}{\Rightarrow} \delta$. If $n \geq 1$ we may write $\delta \stackrel{+}{\Rightarrow} \gamma$.

We call a sequence

$$start \Rightarrow \alpha_1 \Rightarrow \ldots \alpha_n \Rightarrow \gamma$$

a **derivation** of γ.

A string u of elements of T is in the language generated by the grammar if there is a derivation $S \stackrel{*}{\Rightarrow} u$. The elements of the language are called *sentences* of the grammar, and any string α such that $S \stackrel{*}{\Rightarrow} \alpha$ is called a *sentential form* of the grammar.

Example. The following is a derivation of $1 + 2 * 3$ from the grammar above:

$$E \Rightarrow E * D \Rightarrow E + D * D \Rightarrow D + D * D \Rightarrow 1 + D * D \Rightarrow 1 + 2 * D \Rightarrow 1 + 2 * 3.$$

A derivation is *left-most* if the left-most nonterminal in the sentential form is replaced at each derivation step. The above derivation is left-most. A derivation is *right-most* if the right-most nonterminal in the sentential form is replaced at each derivation step. For every left-most derivation there is a corresponding right-most derivation. A nonterminal A is *ambiguous* if there is some string u of terminals such that there are two different left-most derivations of u from A. A grammar is *ambiguous* if is has an ambiguous nonterminal.

4.3 Exercises

1. Describe the languages denoted by the following regular expressions:

   $0(0|1)^*0, \quad ((\varepsilon|0)1^*)^*, \quad (00|11)^*((01|10)(00|11)^*(01|10)(00|11)^*)^*.$

   Write NFAs for each of these regular expressions.

   (Harder) Write a regular expression that denotes the language of all strings of 0's and 1's which contain an even number of 0's and an odd number of 1's.

2. In C++ `floats` are strings which consist of an optional plus or minus sign, followed by one or more digits, followed by an optional sequence containing a decimal point and another non-empty sequence of digits, then followed by an

optional sequence beginning with E, then an optional plus or minus sign, then a non-empty sequence of digits. For example, of the form 1234, –1.2, 34.567, +46.7, 3.333E75, 1004.4444E–3, etc.

Give an NFA which recognises exactly the C++ `floats`, and hence write down a regular expression which describes *floats*.

3. Using the grammar on page above, give two different derivations of the string

```
'if' '(' 'id' '=' 'digit' ')' 'if' '(' 'id' '<' 'digit' ')'
'stop' 'else' 'skip'
```

4. Write a context free grammar for the language `floats` described in Question 2 above.

4.4 Derivation trees

The primary role of the parser is, given a string $u \in T^*$, to determine whether or not u is a sentence in the given language. The parsing techniques which we consider all involve attempting to reconstruct a derivation of the input string u. There are two basic techniques that we consider: top-down and bottom-up. A top-down parser begins with the start symbol and attempts to construct a derivation of the input string from left to right. A bottom-up parser begins with the input string and repeatedly replaces substrings with matching left-hand-sides of production rules until the start symbol is obtained. This reconstructs the parse from right to left. The names top-down and bottom-up reflect the fact that derivations can be represented using trees.

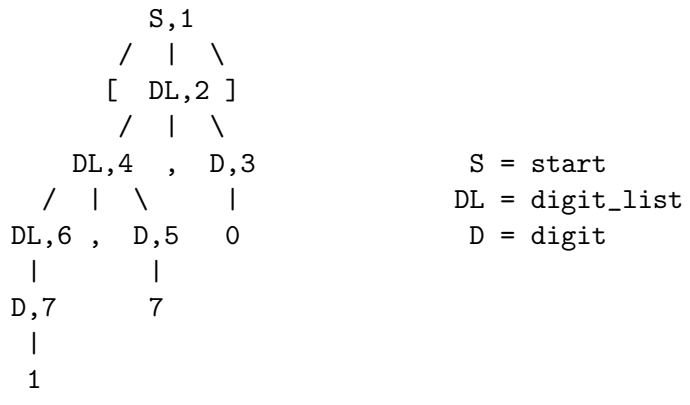
Consider the grammar

```
start ::= [ digit_list ]
digit_list ::= digit | digit_list , digit
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

and the derivation

```
start => [ digit_list ] => [ digit_list , digit ]
      => [ digit_list , 0 ] => [ digit_list , digit , 0 ]
      => [ digit_list , 7 , 0 ] => [ digit , 7 , 0 ] => [ 1 , 7 , 0 ]
```

This derivation can be represented as a tree:



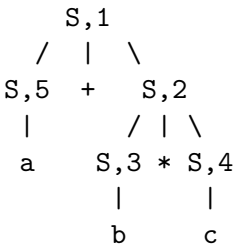


Figure 6 A derivation tree

We can represent derivations as ordered labelled trees. (That is, a tree in which each node is labelled and the children of each interior node have a specified left-right ordering.) Each interior node is labelled with a non-terminal symbol and an integer, and each leaf is labelled with a terminal. The root node is labelled with the start symbol S and the number 1. This node has a child for each symbol in the phrase which immediately follows S in the derivation. The order of the children corresponds to the order of the symbols in the production rule. The node labelled with the non-terminal which is used for the next step in the derivation is labelled 2, and has a child node for each symbol on the RHS of the production rule used. We carry on in this way, labelling the each interior node with the derivation step number and the non-terminal which is expanded at that step. For instance, consider the following grammar and derivation (which are expressed in the theoretical style – non-terminals are capitals and everything else is a terminal):

S ::= S + S | S - S | S ∗ S | 0 | 1 | a | b | c
S ⇒ S+S ⇒ S + S ∗ S ⇒ S + b ∗ S ⇒ S + b ∗ c ⇒ a + b ∗ c

Labelled ordered trees which are constructed from derivations in the above way are called *derivation trees*. The derivation tree describing these steps is shown in Figure 6.

We can read a sentence of the language from the derivation tree by listing the labels of the leaves of the tree from left to right.

Two or more distinct left-most derivation sequences may generate the same sentence and hence sentences can have two or more derivation trees. The following two derivations and corresponding derivation trees also generate the sentence $a + b * c$.

S ⇒ S+S ⇒ a + S ⇒ a + S ∗ S ⇒ a + b ∗ S ⇒ a + b ∗ c
S ⇒ S∗S ⇒ S + S ∗ S ⇒ a + S ∗ S ⇒ a + b ∗ S ⇒ a + b ∗ c

The corresponding trees are shown in Figure 7. If operators deeper in a tree are considered to be more binding than those higher up, the first tree gives the same operator precedence as the original derivation, but the last tree gives addition priority over multiplication.

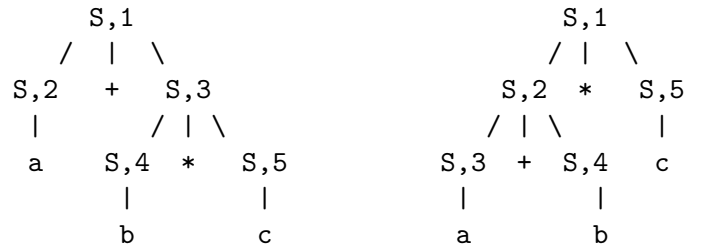


Figure 7 derivations from an ambiguous grammar

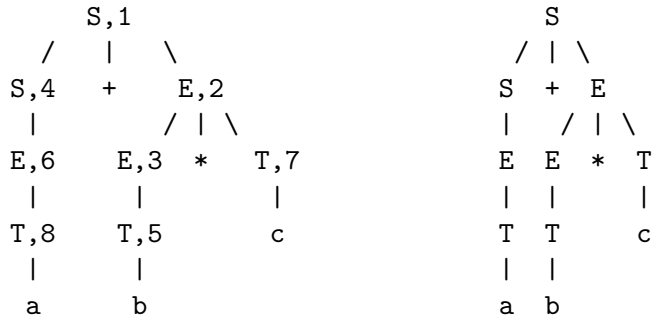


Figure 8 Derivation of arithmetic expressions

All three trees generate the sentence under consideration, and any of them is sufficient to show that the sentence belongs to the language generated by the grammar rule. The rule generating the grammar is 'flat' in the sense that it does not force the derivation trees to have any particular shape. To exclude the derivations which give addition higher priority than multiplication we can modify the grammar rules. The language generated will be identical to the original language, that is we can still generate exactly the same set of sentences, but we can restrict the 'shape' of the derivation trees which can be produced.

For example, the following productions generate the same language.

S ::= S + E | S - E | E

E ::= E ∗ T | T

T ::= 0 | 1 | a | b | c

The following is a derivation of $a + b * c$. The derivation tree is shown in Figure 8.

S ⇒ S+E ⇒ S + E ∗ T ⇒ S + T ∗ T ⇒ E + T ∗ T ⇒ E + b ∗ T
⇒ T + b ∗ T ⇒ T + b ∗ c ⇒ a + b ∗ c

There are still several derivations of the sentence $a + b * c$, for example:

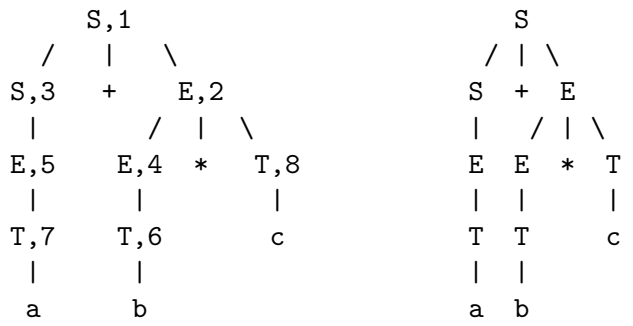


Figure 9 Spurious ambiguity

$$S \Rightarrow S{+}E \Rightarrow S + E * T \Rightarrow E + E * T \Rightarrow E + T * T \Rightarrow T + T * T$$
$$\Rightarrow T + b * T \Rightarrow a + b * T \Rightarrow a + b * c$$

which is shown in Figure 9, but if we remove the numbers from the nodes the derivation trees are indistinguishable. Since subsequent tree traversals are only affected by the *shape* of the tree we have succeeded in specifying the semantics unambiguously by changing the shape of the grammar.

In practice, the order in which the steps in a derivation were carried out is not very important. Thus it is common practice to omit the labels from the nodes of the derivation tree. We shall adopt this convention for most of the rest of this course. However, it is important to remember that a derivation tree without labels will usually correspond to several (essentially equivalent) derivations, but will correspond to exactly one left-most derivation. Thus a grammar is ambiguous if there exist two derivation trees, for the same string, which are distinct even when written without labels.

4.5 Top down parsing

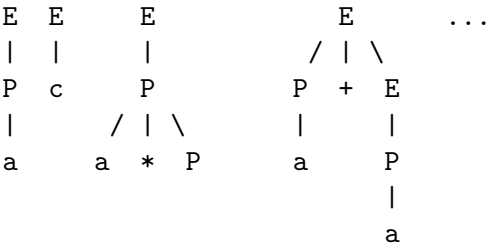
A *top-down parser* constructs parse trees by starting at the root and working down to the leaves. The goal of a top-down parser is to produce, starting from the start symbol, the string of terminals that have been presented to it as input. At any point in the procedure part of the parse tree will have been constructed and the current goal will be to construct the remainder of the tree in such a way that its yield (the string of terminals which label the leaves of the tree) matches the input string. The tree is extended by adding the children of a leaf that is labelled by a non-terminal.

If there is more than one alternate in the production rule for the non-terminal there will be a choice of children to be added. If the parse eventually fails we will need to backtrack to this point and try again with a different choice of children.

4.5.1 Left-most top down parsing

In a left-most top down parser at each stage in the construction we begin by considering the left-most leaf, of the tree so far constructed, that is not a terminal. If no such leaf exists then the parse is complete. Suppose that the first (left-most) child just added to the tree was a non-terminal. In this case we extend the tree using a production rule whose left-hand side is the label of the leaf and begin the process again. If no such rule exists then this attempt has failed. If the first leaf is a terminal then the string obtained by starting with the leftmost leaf of the tree and proceeding to the first leaf that is a non-terminal must be a initial segment of the input string. If this is the case we carry on and begin the process again with the next left-most leaf. If it is not then this attempt has failed. In either case of failure we backtrack to the last point at which we made a choice and try a different choice. If all possible choices at all levels in the tree have been considered and failed then the parse fails and the construction stops. If the parse produces a complete parse tree then there is still failure if the yield is not the input string. If there is no possible backtracking then the parse fails. The parse succeeds if the yield is the input string.

Example: E ::= P | c | P+E, P ::= a | a∗P. Parse: a + a + c.



If the nodes of the derivation tree constructed by a left-most top down parser were numbered in construction order then it would correspond to a left-most derivation.

It is possible to improve the efficiency of a top down parser by guiding the choice of alternate used to extend a non-terminal node in the tree. If b is the current symbol in the input stream and if A is the current (left-most) non-terminal in the tree then we only consider using a production $A ::= \alpha$ to extend the tree if $\alpha \stackrel{*}{\Rightarrow} b\delta$ for some δ. To describe this properly we need to consider the so-called FIRST sets.

4.5.2 FIRST sets

Let G be a grammar and let γ be a string of tokens and non-terminals in G:

FIRST$(\gamma) = \{t \mid t$ is a terminal and $\gamma \stackrel{*}{\Rightarrow} t\alpha$, for some $\alpha\}$.

If $\gamma \Rightarrow^* \varepsilon$ then we also include ε in FIRST(γ). For technical reasons we define FIRST(ε) to be the set $\{\varepsilon\}$.

In the grammar given at the beginning of Section 4.4 we have

FIRST(`start`) = { [} and FIRST(`digit_list`) $=\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

For any token t, FIRST(t) $=\{t\}$, and if $\gamma = X\delta$ then, if $X \stackrel{*}{\Rightarrow} \varepsilon$, FIRST$(\gamma)$ = FIRST$(X) \cup$ FIRST(δ), and FIRST(γ) = FIRST(X) otherwise.

An algorithm for calculating FIRST sets for non-terminals is as follows:

1. Do over all P_i and all alternates α_j of P_i
2. Do over all the nullable leftmost nonterminals X in α_j, calculate FIRST(X) and add FIRST$(X)\backslash\{\epsilon\}$ to FIRST(P_i).
3. If the next symbol in α_j is a terminal, add that terminal to FIRST(P_i)
4. If the next symbol in α_j is a non-terminal X, calculate FIRST(X) and add it to FIRST(P_i).
5. If there is no next symbol add ϵ to FIRST(P_i).

4.5.3 Calculating FIRST sets by hand

If a is a terminal then for any string γ we have

$$\text{FIRST}(a\gamma) \;=\; \{a\}.$$

If A is a non-terminal then for any string γ we have

$$\text{FIRST}(A\gamma) \;=\; (\text{FIRST}(A)\backslash\{\epsilon\}) \;\cup\; \text{FIRST}(\gamma) \qquad \text{if } A \stackrel{*}{\Rightarrow} \epsilon,$$

and

$$\text{FIRST}(A\gamma) \;=\; \text{FIRST}(A) \qquad \text{if } A \;\not\stackrel{*}{\Rightarrow}\; \epsilon,$$

If A is a non-terminal and $A ::= \alpha \mid \beta \mid \gamma$ say, then

$$\text{FIRST}(A) \;=\; \text{FIRST}(\alpha) \;\cup\; \text{FIRST}(\beta) \;\cup\; \text{FIRST}(\gamma).$$

Example

$$S ::= \; aBA \mid BB \mid Bc \qquad A ::= \; Ad \mid d \qquad B ::= \; \epsilon$$

We have

$$\begin{array}{ll}
\text{FIRST}(B) & = \text{FIRST}(\epsilon) = \{\epsilon\} \\
\text{FIRST}(A) & = \text{FIRST}(Ad) \cup \text{FIRST}(d) = \text{FIRST}(A) \cup \{d\} = \{d\} \\
\text{FIRST}(S) & = \text{FIRST}(aBA) \cup \text{FIRST}(BB) \cup \text{FIRST}(Bc) \\
& = \text{FIRST}(a) \cup (\text{FIRST}(B)\backslash\{\epsilon\}) \cup \text{FIRST}(B) \cup (\text{FIRST}(B)\backslash\{\epsilon\}) \cup \text{FIRST}(c) \\
& = \{a, \epsilon, c\}
\end{array}$$

A left-most top down parser that uses the first sets to guide the parse in this way is an *LL parser*, because it reads the input from left to right and constructs a left-most derivation. It is possible to use more than one symbol of lookahead so we sometimes use the term LL(1) parser to indicate that one symbol of lookahead is being used.

4.6 Grammars which admit top down LL(1) parsers

We can view a parser as taking input a string and either not terminating, or terminating and returning 'success' or 'failure'.

We say that a grammar is *compatible* with a parsing technique if, given a parser which uses the technique, whenever the parser terminates and returns success on an input string u then u is in the language of the grammar.

We say that a grammar *admits* a parsing technique if it is compatible with it and if, given a parser which uses the technique, for every input string the parser terminates and returns success if the string is in the language of the grammar.

All of the parsing techniques which we shall consider are compatible with every context-free grammar. However, top down LL parsers can fail to terminate for some grammars, and all of the techniques we consider either require back-tracking or can terminate in failure on some strings in the language of some grammars.

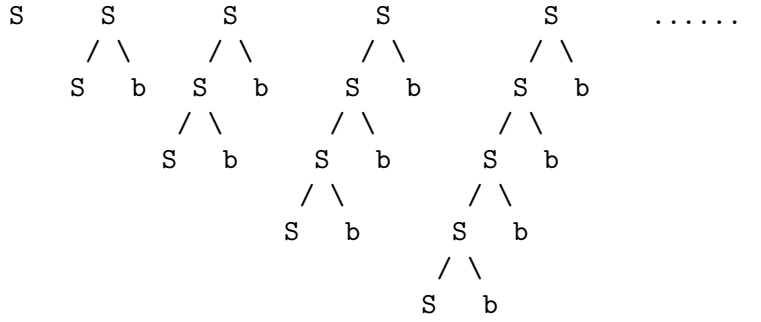
We begin by looking at grammar properties which can prevent a parser from terminating, and then we shall look at grammar properties which remove the need for back-tracking.

4.6.1 Left recursion

Consider the grammar

$$S ::= Sb \mid a$$

and suppose that we attempt to use a top down LL parser to parse the string *abbb*. Since at each stage the left-most non-terminal is extended the tree constructing process may never terminate.



The procedure recursively calls itself from the left forever. This prompts the following definition:

A grammar is *left recursive* if it has a nonterminal A such that there is a derivation sequence $A \stackrel{+}{\Rightarrow} A\alpha$ for some string α. A production shows immediate left recursion if $A ::= A\alpha$.

There is an algorithm which removes left recursion from a grammar provided that it has no cycles (derivations of the form $A \stackrel{+}{\Rightarrow} A$) or ϵ-productions. Since there are also algorithms which remove cycles and ϵ-productions from grammars, it is always possible to remove the left recursion from a grammar.

4.6.2 Left recursion removal algorithm

We begin by describing an algorithm which removes immediate left recursion and then use this as part of a general left recursion removal procedure.

Collect together all productions of A into a single production rule of the form:

$$A ::= A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n.$$

where no β_i begins with A.

This may then be replaced with the following rules:

$$A ::= \beta_1 B \mid \beta_2 B \cdots \mid \beta_n B.$$

$$B ::= \alpha_1 B \mid \alpha_2 B \cdots \mid \alpha_n B \mid \epsilon$$

where B is a non-terminal.

We now give the general left recursion removal algorithm.

Arrange the nonterminals in some order, say $A_1, A_2, \ldots A_n$.

For i from 1 to n do
  For j from 1 to $i-1$ do
  replace each production of the form
    A_i ::= $A_j\gamma$ by
    A_i ::= $\delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$
    where A_j ::= $\delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are the current A_j productions.
  eliminate the immediate left recursion amongst the A_i productions.

Theorem The grammar obtained using the above algorithm is equivalent to (generates the same language as) the original grammar.

Example Remove the left recursion from the grammar

$$S ::= aBA \mid SB \mid Bc \qquad A ::= Sa \mid d \qquad B ::= Ad$$

Order the nonterminals S, B, A.
Remove the immediate left recursion from S

$$S ::= aBAM \mid BcM \qquad M ::= BM \mid \epsilon$$

The rules for B have no immediate left instance of S or B so there is nothing to do.
Substitute for initial instances of S in the rules for A

$$A ::= aBAMa \mid BcMa \mid d$$

Then substitute for initial instances of B

$$A ::= aBAMa \mid AdcMa \mid d$$

Remove the immediate left recursion from A giving the final grammar

$$S ::= aBAM \mid BcM \qquad M ::= BM \mid \epsilon \qquad B ::= Ad$$

$$A ::= aBAMaN \mid dN \qquad N ::= dcMaN \mid \epsilon$$

4.6.3 Left factored grammars

If a parser has to back-track then the processing overheads are high, and often unacceptable. If it is to guarantee to find a derivation, an LL parser will have to back-track if it selects the wrong alternate from a production rule. We choose an alternate only if the current input symbol is in the FIRST set of that alternate. Thus, if each terminal is in the FIRST set of at most one alternate in each production rule then we can tell which alternate to use at each step in the construction of the derivation tree. This prompts the following definition:

A grammar is *left factored* if the FIRST sets of any two alternates in a production rule have no elements in common, i.e. if $A ::= \alpha|\beta$ then

$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$.

Sometimes it is possible to transform a grammar which is not left factored in to one which is by adding a new non-terminal and associated production rule. We can replace a rule of the form

$$A ::= \gamma\alpha|\gamma\beta$$

with the rules

$$A ::= \gamma B \qquad B ::= \alpha|\beta$$

4.6.4 Follow determinism

There is another reason for possible choice (non-determinism) in an LL parser which can create the need for back-tracking. You can think of this as needing to know when to stop matching a portion of the input to a particular non-terminal.

Consider the grammar

$$S ::= Aa \qquad A ::= a|\epsilon$$

If we are asked to parse the string aa then we need to use the alternate $A ::= a$, but if we are asked to parse the string a we need to use the alternate $A ::= \epsilon$, and we can't tell which alternate to use just be looking at the current input symbol because in both cases it is a. The problem here is that the input symbol a can be both the start of a string generated by A and the start of the rest of a string following an occurrence of A. The set of terminals which can begin the rest of a string following an occurrence of a non-terminal are thus of interest for guiding a parser.

For a non-terminal A we define

$\text{FOLLOW}(A) = \{t \mid t \text{ is a terminal and } S \stackrel{*}{\Rightarrow} \beta A t \alpha, \text{ for some } \beta, \alpha\}$.

If there is a derivation of the form $S \stackrel{*}{\Rightarrow} \beta A$ then we also add a special end-of-file symbol, usually written \$, to $\text{FOLLOW}(A)$. So in particular, $\$ \in \text{FOLLOW}(S)$.

Calculating FOLLOW sets by hand

For each non-terminal A and for each rule $B ::= \alpha A \beta$ we define a local follow set of A as

$$FW_{B::=\alpha A\beta}(A)$$

to be FIRST(β)\{ϵ}. If $\beta \stackrel{*}{\Rightarrow} \epsilon$ then we also add FOLLOW(B) to $FW_{B::=\alpha A\beta}(A)$.

The FOLLOW(A) is the union of all the local follow sets for A, with \$ added if $A = S$.

Example

$$S ::= aBA \mid SB \mid Bc \qquad A ::= Aa \mid d \qquad B ::= d$$

We have

$$\text{FOLLOW}(S) = \{\$\} \cup FW_{S::=SB}(S) = \{\$\} \cup \text{FIRST}(B) = \{\$, d\}$$

$$\text{FOLLOW}(A) = FW_{S::=aBA}(A) \cup FW_{A::=Aa}(A) = \text{FOLLOW}(S) \cup \{a\} = \{\$, d, a\}$$

$$\text{FOLLOW}(B) = FW_{S::=aBA}(B) \cup FW_{S::=SB}(B) \cup FW_{S::=Bc}(B) = \{d\} \cup \{\$, d\} \cup \{c\}$$

A grammar is said to be *follow determined* if for any non-terminal A, strings γ, δ and any terminal a,

$$A \stackrel{*}{\Rightarrow} \gamma \text{ and } A \stackrel{*}{\Rightarrow} \gamma a \delta \text{ imply that } a \notin \text{FOLLOW}(A).$$

It turns out that if a grammar is left factored then the following property is enough to guarantee that it is also follow determined: For any non-terminal A, if $A \stackrel{*}{\Rightarrow} \epsilon$ then for all β such that $A ::= \beta$,

$$\text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \emptyset.$$

4.6.5 LL(1) grammars

Grammars which admit non-back-tracking top down LL(1) parsers are precisely the ones which are left factored, follow determined and have no left recursion. Thus we have the following definition:

A context-free grammar is LL(1) if for all non-terminals A and productions $A ::= \alpha|\beta$ we have

1. FIRST(α) $\cap$ FIRST(β) = $\emptyset$
2. If $A \stackrel{*}{\Rightarrow} \epsilon$ then FIRST(A)$\cap$FOLLOW(A) =$\emptyset$.

4.7 Top down LL parsing and recursive descent

We now consider an algorithm for implementing a top down LL parser. It is possible store some processed version of the grammar in a table and use a function to traverse the table, constructing portions of the tree according to the table entry. This kind of parser is called a *table driven* parser and we shall look at these in detail in Section 5.

An alternative approach, which has a particular affinity with top down LL parsing, is to view the grammar as a form of parsing schedule comprised of many individual sub-parsers, one *per* nonterminal. Each sub-parser is a function that attempts to find a match against one of the alternate productions

within that nonterminal's rule. Instances of nonterminals on the righthand side of a production are matched by calling the relevant sub-parser function: it is clear therefore that the functions might be called recursively. A parser constructed in this way is called a *recursive descent* (RD) parser and RD parsers for LL(1) grammars are so easy to understand that they amenable to manual construction. As such they are very popular in practice, and some languages (especially Pascal) were designed to make them easy to parse using recursive descent.

For example, we suppose we have functions `gnt()` which calls the lexer and returns the next token, and `error()` which terminates the program and prints a suitable error message. We write parse functions, `parseA()` for each non-terminal A, and a main function which calls the parse function for the start symbol.

```
main() {
   x = gnt();
   parseS();
   if x == $ return success else return failure; }
```

Each parse function has a nested `if` statement which tests to see if the current input symbol is in the FIRST set of each alternate. For each alternate we consider each symbol. If it is a non-terminal, A, we call the `parseA()`. If it is a terminal we match against the input symbol and either call `gnt()` or `error()`.

$$S ::= T\ a \mid b\ c \mid a\ T\ S \qquad T ::= c\ T \mid d\ T \mid \epsilon$$

```
parseS() {
if (x==c or x==d or x==a) {
   parseT();
   if (x==a) x=gtn() else error(); }
else if (x==b) {
        if(x==b) x=gnt() else error();
        if(x==c) x=gnt() else error(); }
     else if(x==a) {
               if(x==a) x=gnt() else error();
               parseT();
               parseS(); }
          else error(); }
```

We match an ϵ alternate by simply returning rather than calling `error()` if no alternate is selected.

```
parseT() {
if (x==c) {
   if (x==c) x=gtn() else error(); }
   parseT();
else if (x==d) {
        if(x==d) x=gnt() else error();
        parseT(); }
```

The general method for constructing an RD parser from a grammar is as follows.

Construct a main function that declares a global input variable `x` say, and then calls a lexer function, `gnt()` say, to initialise `x`. Then call the parse function, `parseS()`, for the start symbol `S`. Finally check to see if the value of `x` is the end of string symbol and issue a success or failure message.

For each nonterminal `A` construct a parse function `parseA()` which has an outer **if** statement that tests the value of `x` against the FIRST set of each alternate of `A` and FOLLOW(A) if the alternate is nullable. For each non-ϵ alternate `X1 X2 ... Xn` and for each `Xi` in the alternate add a line to `parseA()` as follows. If `Xi` is a nonterminal add a call to `parseXi()`. If `Xi` is a terminal, if it equals the current value of `x`, call `gnt()` and assign the result to `x`, otherwise issue an error message and terminate the parser. Finally, if there is no rule $A ::= \epsilon$, the last branch of the **if** statement should issue an error message and terminate the parser.

In Section 4.9 we will describe a real parser generator, **rdp**, which can read a grammar, check that it is LL(1) and output a recursive descent parser written in C. The input grammars for **rdp** are written using a slightly different form of BNF called *extended BNF* (EBNF) which we now describe.

4.8 EBNF

So far we have only considered defining grammars using BNF. However, some patterns which occur in production rules are so common that it is useful to define shorthand notations for them.

- ◇ parentheses (...) which create a sub-production (an unnamed production),
- ◇ brackets [...] as shorthand for 'optional' or 'zero or one occurrence',
- ◇ braces {...} as shorthand for 'zero or more occurrences'.
- ◇ angle brackets <...> as shorthand for 'one or more occurrences'

Other notations that you might meet include

- ◇ (...)? for [...],
- ◇ (...)* (or *Kleene closure*) for {...} and
- ◇ (...)+ (or *positive closure*) for <...>.

A common construction is the *delimited list*. For instance, the array subscript example may be rewritten as

```
start ::= '[' digit { ',' digit } ']'
```

(Here the terminals have been quoted.) A non-standard notation for this is as follows:

```
start ::= '[' (digit)@',' ']'
```

The body of the expression (...) repeats after inserting a comma, as often as required.

From a parsing perspective, `{',' digit}` reads like a WHILE DO loop and `digit @ ','` like a DO WHILE or REPEAT UNTIL loop

A further occasional requirement is to specify upper and lower limits on an iteration. For instance, in some macro languages an upper limit of nine or ten parameters is enforced. Variable name lengths are also commonly restricted. The list construct has the following syntax:

```
list::= ( alt ) [ INTEGER @ INTEGER token ]
```

The first optional integer is the low count limit and the second integer the high limit. So, for instance, a parameter list that must be between two and ten elements long is written

```
params ::= '(' ( ID ) 2@9',' ')'
```

A Pascal-style identifier with a ten character limit is written:

```
p_id ::= alpha ( alpha | digit ) 0@9 #
```

where `#` represents the empty string.

Our list construct subsumes all the other EBNF constructs. Firstly, we allow a high limit of zero to represent 'no high limit'. Secondly, the default values of the '@' attributes are:

- ◇ high limit zero, and
- ◇ low limit one

We have the following synonyms:

{...}	(...)*	(...)0@0#
<...>	(...)+	(...)1@0#
[...]	(...)?	(...)0@1#
(...)	(...)	(...)1@1#

As an example we give the grammar for `C--`, a subset of ANSI C. Any valid `C--` program will be accepted by an ANSI compiler. The language differs from C in that it has no pointer operations, no user defined types, and no SWITCH, GOTO, BREAK or CONTINUE statements.

```
text ::= { type_name ID ( '(' (type_name ID)@',' ')' [ block ] |
                          ['=' conditional ]
                          [ ',' (ID ['=' conditional])@',' ] ';'
                        )
         } .

block ::= '{' { declaration } { statement } '}'.
```

```
declaration ::= type_name ( ID ( ['[' conditional ']'] |
                                 ['=' conditional] )
                          )@',' ';'.

statement ::=
   'while' '(' expression ')' statement                        |
   'do' statement 'while' '(' expression ')' ';'               |
   'if' '(' expression ')'  statement [ 'else' statement ]     |
   'for' '('expression';'expression';'expression')' statement  |
   'return' [ expression ] ';'                                 |
   expression ';'                                              |
   block.

type_name ::= 'char' | 'float' | 'int' | 'unsigned' | 'void'.

expression ::= assignment @ ',' .
assignment ::= conditional{( '=' | '*=' | '/=' | '=' | '+=' | '-='
                           | '<<=' | '>>=' | '&=' | '^=' | '|='
                           ) assignment
                          }.
conditional ::= lor [ '?' expression ':' conditional ].

lor ::= land { '||' land }.
land ::= bor { '&&' bor }.
bor ::= bexor { '|' bexor }.
bexor ::= band { '^' band }.
band ::= equality { '&' equality }.
equality ::= relation { ('==' | '!=') relation }.
relation ::= shift { ('<' | '<=' | '>' | '>=') shift}.
shift ::= add { ('<<' | '>>') add}.
add ::= mul { ('+' | '-') mul}.
mul ::= prefix { ('*' | '/' | '%') prefix}.
prefix ::=
    {'++' | '--' | '+' | '-' | '~' | '!' | '*' | '&'} postfix.
postfix ::= primary [ '++' | '--' ] .
primary ::= ID  { '[' expression ']' | '(' expression ')' }|
            INTEGER | REAL |
            STRING_ESC('"' '\\') | '(' expression ')' |
            STRING('\'') .
```

A string (program) in the language C-- is:

```
int a, b = 2,c;
float f = 3.2;
char cc = 'c';

void empty(int z) { }

float z(int a, char c) {
  int d;

  d = a+3 << 2 && 0x56;
```

```
  while (3>a) {
    empty(a);
    a++;
  }
  return 3.6;
}
```

4.9 rdp

`rdp` is a system for implementing language processors. Compilers, assemblers and interpreters may all be written in the `rdp` source language (an extended Backus-Naur Form) and then processed by the `rdp` command to produce a program written in ANSI C which may then be compiled and run. `rdp` takes as input an LL(1) grammar and produces a recursive descent based parser for the language defined by that grammar.

```
(***** functn1.bnf *****)

S ::= INTEGER E .
E ::= '+' S | '-' S | ';' .

static void E(void){
  if (scan_test( + )) {
    scan_();
    S(); }
  else
    if (scan_test( - )) {
      scan_();
      S(); }
    else
      if (scan_test( ; )) { scan_(); }
      else { /* report error in input */ }
}

void S(void){
  scan_test( INTEGER );
  scan_();
  E();
}

int main( ... ) {
  ...
  scan_();
  S();         /* call parser at top level */
  ...
}
```

4.9.1 Acceptable languages

`rdp` generates parsers that work using recursive descent. It requires grammars to be LL(1) (or something very close to LL(1)), that is they must be unambiguously parsable using a single token of lookahead and there must be no left

recursive rules. This is not a significant constraint for modern languages like Ada and Pascal which were to some extent designed with a view to easy parsing. The main advantages of recursive descent parsing are

- ◇ fast (linear time) parsing,
- ◇ the availability of standardised error recovery mechanisms and
- ◇ the straightforward one-to-one relationship between the code in the parser and the productions in the source EBNF. This makes it straightforward to debug the grammar, because a C language debugger may be used to step through the parser functions and, equivalently, to step through the grammar productions.

If the grammar you present to `rdp` is not LL(1), `rdp` issues detailed diagnostics explaining which tokens and productions are giving the trouble. It is possible to write algorithms that translate some non-LL(1) grammars automatically to LL(1) form, but `rdp` does not attempt to perform any such transformations because that would produce an obscure parser that was no longer directly related to the input grammar.

`rdp` is itself a language processor, and the `rdp` source language is LL(1). In fact `rdp` is written in itself — an early version of the system was hand written and later versions developed from a grammar written in terms of itself.

4.9.2 System flow

The steps involved in producing a new language processor for a mythical new language `myth` with `rdp` are

1. Create a file `myth.bnf` containing an EBNF description of the `myth` language. Decorate the grammar with attributes and C-language fragments describing semantic actions. By convention, large semantic routines are kept in a file called the *auxiliary* file with a name like `myth_aux.c`.
2. Process `myth.bnf` with `rdp` to produce `myth.c`.
3. Compile `myth.c` and `myth_aux.c` with an ANSI C compiler.
4. Link the `myth` object file with the `rdp_supp` modules and any other required semantic routines.
5. Run the resulting executable on `myth.str`, a test program for the `myth` language.

This process is illustrated in Figure 10.

4.9.3 An example – the `mini` language

This section concerns the construction of a parser for a tiny language called `mini`. The language includes variable declaration, assignment, the basic arithmetic operators and a `print` procedure that can output a mix of variables and strings. The grammar for language `mini` is shown in Figure 11.

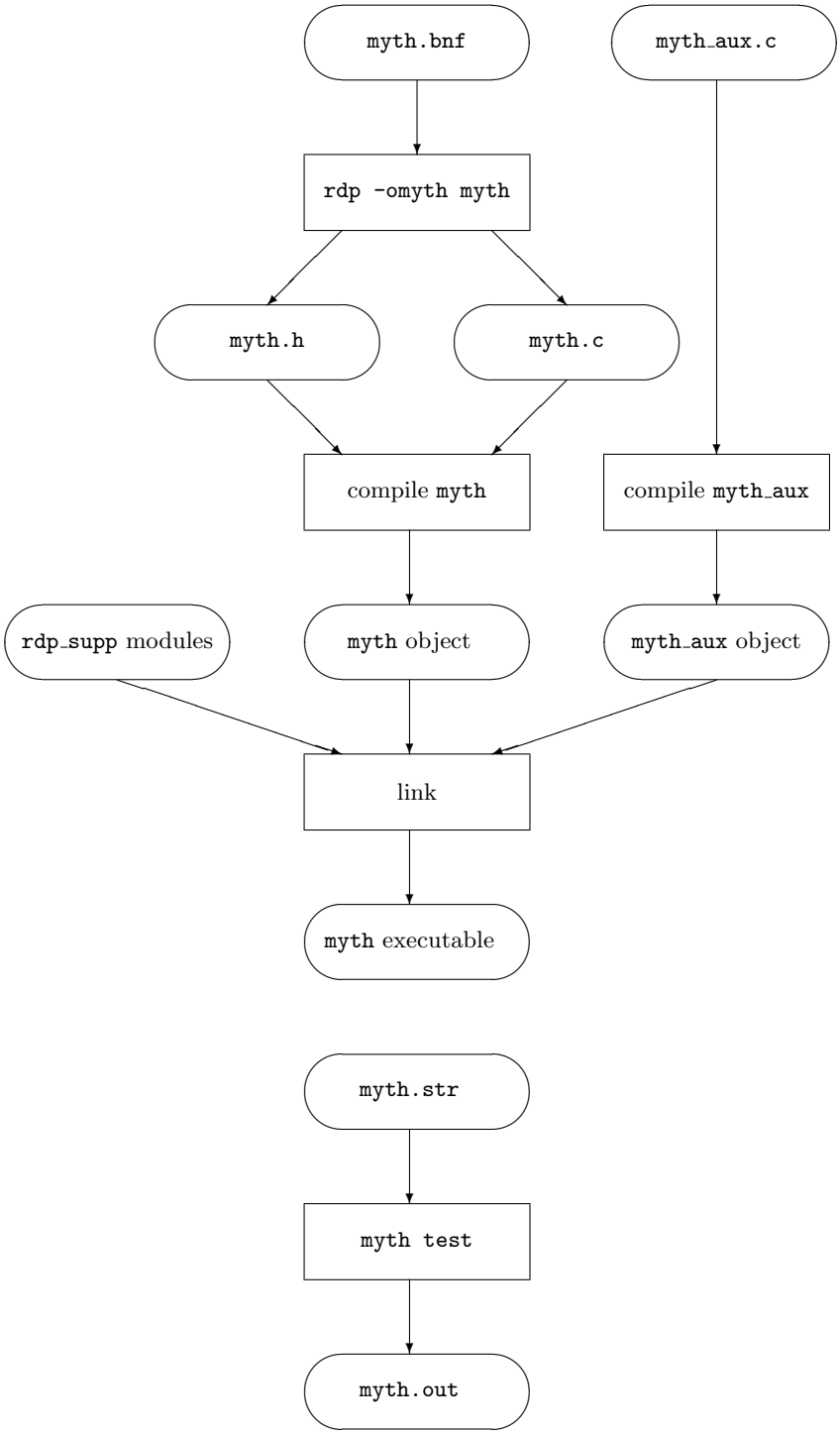


Figure 10 `rdp` design flow

```
program   ::= { [var_dec | statement ] ';' }.

var_dec   ::= 'int' ( ID [ '=' e1 ] )@',' .

statement ::= ID '=' e1 |
              'print' '(' ( e1 | String )@',' ')'.

e1 ::= e2 {'+' e2 |    (* Add *)
           '-' e2 }.   (* Subtract *)

e2 ::= e3 {'*' e3 |    (* Multiply *)
           '/' e3 }.   (* Divide *)

e3 ::= '+' e4 |        (* Posite *)
       '-' e4 |        (* Negate *)
       e4.

e4 ::= ID |            (* Variable *)
       INTEGER |       (* Numeric literal *)
       '(' e1 ')'.     (* Parenthesised expression *)

comment ::= COMMENT_NEST('(*' '*)'). (* Comments *)

String ::= STRING_ESC('"' '\\').        (* Strings for print *)
```

Figure 11 The `mini` grammar

Identifiers for production names follow C style rules: that is they must start with a letter or underscore and can contain digits, letters or underscores. The only limit on identifier length is the available text memory. `rdp` checks that you have not used an identifier name that clashes with a C reserved word which would of course cause surprising compile time errors.

Literal tokens are enclosed in single quotes. `rdp` understands a family of built-in tokens for parameterised lexemes which includes `ID` for an alphanumeric identifier, `INTEGER` for a C style unsigned integer (including hexadecimal numbers), `EOLN` for the end of line token and `REAL` for a C style real number.

`rdp` also understands a set of paramaterisable tokens used to describe strings and comments. In the `mini` grammar `COMMENT_NEST` defines comment brackets `(*` and `*)` which may be nested. The `STRING_ESC` primitive specifies a C-style string literal delimited by double quote characters (`"..."`) with C-style escape sequences introduced by the \ character. There is another string primitive that implements Pascal style string literals.

EBNF files can also contain directives like `TITLE` and `SUFFIX`. These directives are used to control some aspects of the resulting parser. `TITLE` defines a program title that will appear if the user of the `mini` parser asks for verbose operation or makes an error. The `SUFFIX` directive declares a default file type for the files that `mini` will parse.

4.9.4 Building a `mini` parser

The grammar in Figure 11 can be submitted on its own to `rdp` which will check it for non-LL(1) constructs and then construct a recursive descent parser written in ANSI C. When the parser is compiled, it may be run on a `mini` source file at which point any syntax errors in the `mini` file will be reported.

On a Unix system with gcc, the following commands build and run the `mini` parser:

```
rdp -omini mini
gcc -o mini mini.c
mini mini.str
```

The following, `mini.str`, is a suitable mini test file.

```
int a=3+4, b=1;
print("a is ", a, "\n");
b=a*2;
print("b is ", b, ", -b is ", -b, "\n");
int z = a;
print(z, "\n");

(* End of mini.str *)
```

4.10 Strategies For Dealing With Ambiguity

If a grammar does not satisfy the conditions for a given parsing technique then the parser will encounter a choice of action when using that grammar, and may incorrectly reject a string if it makes the wrong choice. However, we may want to use the parsing technique anyway, and in this case we build the parser so that it makes the choice according to particular strategies.

For recursive descent parsers, one strategy is the *first match strategy*. With this stragety the parser chooses the first alternate in the rule whose FIRST set contains the current input symbol. The nested 'if' statement in our rd parsers automatically implements this approach.

Another strategy is the *longest match strategy*. With this strategy, if it is possible to match two strings γ and $\gamma a\beta$ to a non-terminal A then, when the current input is a we choose to continue to try to match $\gamma a\beta$ rather than matching γ.

If a grammar is left factored then this choice will only arise when $\gamma = \epsilon$. In this case a recursive descent parser will implement longest match as part of its first match strategy because the ϵ-rule is the last choice, executed only if none of the alternates have the current input symbol in their FIRST sets. For `rdp`, if you run `rdp` with the flag `-F` this will force `rdp` to build a parser even if the input grammar is not LL(1). The parser will use the first and longest match strategies discussed here.

Given the rules

$$S ::= \ 'if'\ B\ 'then'\ S\ E \mid 'a' \quad E ::= \ 'else'\ S \mid \epsilon$$

a longest match rd parser will correctly resolve the well-known 'if-then-else' ambiguity.

These strategies will only work correctly on ambiguities. If the nondeterminism is not an ambiguity then the application of these strategies will cause some inputs to be incorrectly rejected by the parser. Thus they should be used with extreme caution and only when the user fully understands their implications.

5 Syntax analysis II – bottom-up parsers

There is a large class of grammars which can't be parsed using the LL top-down technique without the need for back-tracking. Some of these grammars can be modified so that LL techniques can be used efficiently, but there exist languages for which it is impossible to write an LL(1) grammar.

In this section we consider bottom-up parsing techniques (building the parse tree from the leaves up). We will look at LR(0) and SLR(1) parsing, which admit successively larger classes of grammars. The 'most powerful' technique, LR(1), can handle most grammars which arise in practice, although we shall not focus on this in this course.

We are looking for algorithms which take as input a string $X_1X_2\ldots X_n$, say, of terminals *and* non-terminals and which produce as output another string $Y_1Y_2\ldots Y_m$, if one exists, such that

$$S \stackrel{*}{\Rightarrow} Y_1Y_2\ldots Y_m \Rightarrow X_1X_2\ldots X_n.$$

Such an algorithm is applied to an input string, then successively to its own output with the aim of obtaining the string containing just the start symbol, `S`.

5.1 State machines for finding derivations

Suppose that we have an input string $X_1X_2\ldots X_n$ and that we are trying to construct the previous step $Y_1Y_2\ldots Y_m$ in the derivation.

We cannot expect to know for certain whether the string $X_1X_2\ldots X_n$ can be derived from S because determining whether strings can be derived from the start symbol is the point of the whole process. If we could always tell at any stage, then we wouldn't need the process at all.

What we can do is check that the portion of the input that we read is an initial segment of some sentential form. If not then we can terminate the process because the parse has failed. Next we describe how to construct an NFA which works as follows:

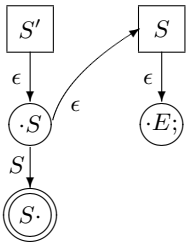
- ◇ It reads part of the input string $X_1\ldots X_i$, say, and then stops.
- ◇ When it stops it either reports an error, or there is some string α such that $S\stackrel{*}{\Rightarrow}X_1\ldots X_i\alpha$ and there is a production $Z ::= X_jX_{j+1}\ldots X_i$. In the latter case, the string $X_1\ldots X_{j-1}ZX_{i+1}\ldots X_n$ is returned. Replacing the right hand side of a rule with its left hand side in this way is called a *reduction*.

We shall illustrate the construction and operation of the NFA using the following grammar which has a special augmented start rule $S' ::= S$.

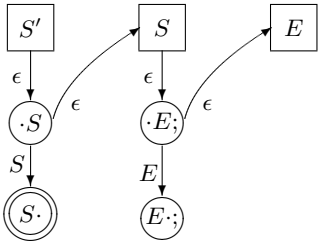
$$\begin{array}{lcl} S' & ::= & S \\ S & ::= & E; \\ E & ::= & E+T \mid T \\ T & ::= & 0 \mid 1 \end{array}$$

The start node of our NFA is a node labelled with S' the augmented start symbol. We are looking to construct a string just containing S and we describe this state using the notation $\cdot S$. So we create a node labelled $\cdot S$ and an arrow labelled ϵ to it from the start node. If the input string is just S then we have a complete, single step derivation and we can terminate and accept the string. We achieve this by creating an accepting node, labelled $S\cdot$ to indicate that we have seen the string S, and an arrow labelled S to this node.

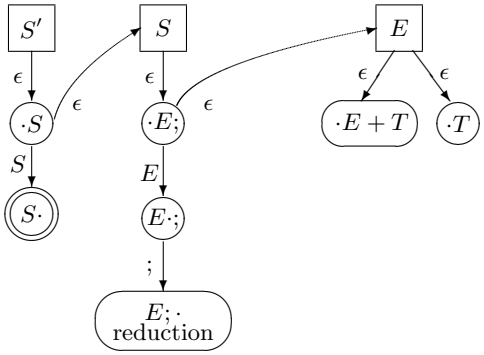
If the input string is not S then we are looking to construct it, and this is indicated by the node labelled $\cdot S$. To construct S we need to find one of the alternates on the right-hand-side of a production rule for S. We create a new special square node labelled S which has a child node for each alternate of the rule for S (in this case E;). We put a dot in front of each of the alternates to indicate that we are now looking for that string. Since the move from S to one of it's alternates doesn't consume any of the input string the arrow between these nodes is labelled ϵ.



So now we are looking for E. If we read the next symbol of the input and it is E then we can move on and look for the next symbol, in this case a semicolon. Otherwise we need to look to construct an E. We build this into the NFA by making a new square node labelled E. Thus we get

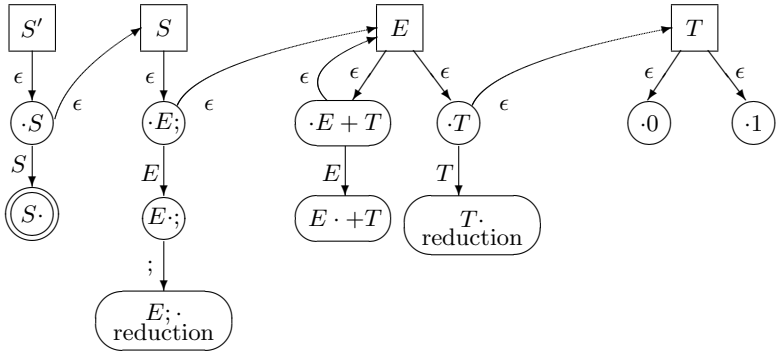


To construct E then, as for S, we need to look for some alternate in the production rule for E, so we add new nodes and epsilon arrows as for S. If we are looking for a ';', and we find one by reading the next input symbol, then we move into a terminating (reduction) state.

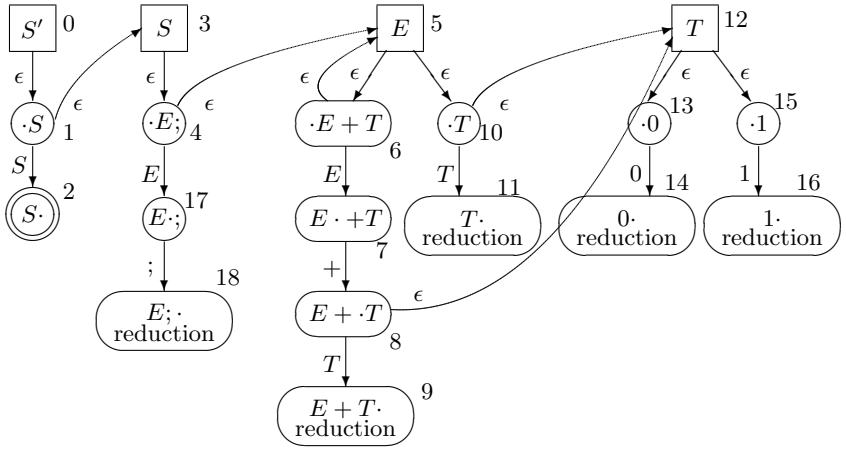


Once the NFA is in a reduction state it will have read from the input an alternate from the production rule of the non-terminal in the nearest square node back. In this case the node labelled S. Thus we can replace the alternate with the non-terminal and return the resulting string.

In the case where we are looking to construct E we find we need to construct either $E + T$ or T. We may find either of these by reading the next input symbol, or we may look to try to construct one of them. The decision to try to construct E is represented by an ϵ-arrow back to the node labelled E, and to try to construct T we create a new square node labelled T and proceed as for the S and E nodes.



We carry on this construction process until all the branches of the NFA terminate in reduction states. The complete NFA for the above grammar is



5.2 Using the state machine to parse

Given an input string $X_1X_2 \ldots X_m$ we begin in the start state of the NFA. Then at each stage we either move along an ε arrow into another state or we read the next input symbol and move into a new state along an arrow labelled with that symbol, if one exists. If no moves are possible then the parse has failed. If the NFA moves into a reduction state the string labelling this state will occur in the input string. Replace this string with the non-terminal in the nearest square node back, and return the result.

Suppose that we are attempting to parse the string $0 + 1;$ in the above grammar, and that we have so far constructed $E + 1;$. Thus we have the derivation steps $E + 1; \Rightarrow T + 1; \Rightarrow 0 + 1;$. To construct the next step, starting in state 0 we run the NFA on $E + 1;$.

```
state      input              action
  0       ^E + 1 ;          move to state 1
  1       ^E + 1 ;          move to state 3
  3       ^E + 1 ;          move to state 4
  4       ^E + 1 ;          move to state 5
  5       ^E + 1 ;          move to state 6
  6       ^E + 1 ;          read input symbol
  7        E^+ 1 ;          read input symbol
  8        E +^1 ;          move to state 12
  12       E +^1 ;          move to state 15
  15       E +^1 ;          read input symbol
  16       E + 1^;          replace 1 by T, return string E + T ;
```

We now have the steps $E+T; \Rightarrow E+1; \Rightarrow T+1; \Rightarrow 0+1;$ and we can run the NFA again on the input $E+T;$.

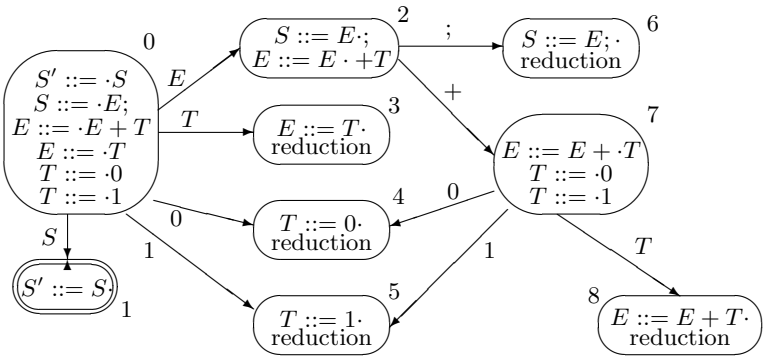
5.3 DFAs and LR(0) parse tables

The problem with the method we have described so far is that the NFA is non-deterministic. If we want to use the NFA directly on a sequential computer, then whenever we encounter non-determinism we must make an arbitrary decision as to which transition to pursue. If it later turns out not to lead to a reduction state, then we must backtrack and try the next option. This backtracking can lead to very inefficient matching, however, we can use the subset construction to reduce an NFA to an equivalent DFA which can then be traversed in linear time.

5.3.1 DFAs from grammars via the subset construction

The states of the DFA are labelled with all of the productions which were composed in the subset construction. Since we lose the square nodes we need to add the left-hand-sides of the productions as well.

Applying the subset construction to the NFA from the previous section gives



The process of constructing the NFA then using the subset construction to get the DFA is rather long winded. We give some definitions and then an algorithm to construct the DFA directly.

A production $A ::= \gamma \cdot \alpha$ which has a 'dot' somewhere on its right hand side is called an *LR(0)-item*.

If P is a set of items the *closure* of P, cl(P), is the smallest set which contains P and all items of the form $B ::= \cdot\beta$ where there is item of the form $A ::= \gamma \cdot B\alpha$ in P.

If P is a set of items and X is a terminal or non-terminal then P_X is the set of all items $A ::= \gamma X \cdot \alpha$ such that $A ::= \gamma \cdot X\alpha$ is in P. We define $move(P, X) = cl(P_X)$.

5.3.2 Algorithm to directly construct an LR(0) DFA

- ◇ If S is the start symbol of the grammar and the production rule for S is $S ::= \alpha_1 | \ldots | \alpha_n$, construct the start state

$$0 = cl(\{S' ::= \cdot S\}) = cl(\{S' ::= \cdot S, S ::= \cdot\alpha_1, \ldots, S ::= \cdot\alpha_n\})$$

- ◇ For each grammar symbol X, form the set of items 0_X and then the closure $cl(0_X)$. If this set is not empty and has not been constructed before it is a new state in the DFA. We draw an arrow labelled X from state 0 to the state 0_X whenever $cl(0_X)$ is not empty.

- ◇ For each new state n constructed and each grammar symbol X, form the set $cl(n_X)$ and add an arrow labelled X from state n to state $cl(n_X)$ as before. When no new states are constructed the DFA is complete.

- ◇ Let $1 = cl(O_S)$ and mark this state as the accepting state.

- ◇ Reduction (accepting) states are the states which contain items of the form $A ::= \alpha\cdot$

5.3.3 Parsing with a DFA

The parsing algorithm based on a DFA is slightly more straightforward than the one based on the NFA. At every stage in the parse we read in the next input symbol and move to the corresponding state if there is one. If there is no state

then the parse has failed. If we move into a reduction state then we replace the identified right-hand-side of a production by the corresponding left-hand-side. If we are in the acceptng state and the input symbol is $, success is reported.

It is usually more convenient to write the DFA as a table whose rows are states, whose columns are input symbols and whose entries are the states to move into from the given state if the given symbol is read. Such tables are called *LR(0) parse tables*.

5.3.4 LR(0) grammars

A grammar is LR(0) if its LR(0) parse table does not contain any multiple entries, or any rows with both a reduction and another non-empty entry.

The LR(0) parse table for the example we have been considering is

	0	1	S	E	T	+	;	reduce by	
0	4	5	1	2	3	-	-	-	
1	-	-	-	-	-	-	-	-	report success
2	-	-	-	-	-	7	6	-	
3	-	-	-	-	-	-	-	$E ::= T$	
4	-	-	-	-	-	-	-	$T ::= 0$	
5	-	-	-	-	-	-	-	$T ::= 1$	
6	-	-	-	-	-	-	-	$S ::= E;$	
7	4	5	-	-	8	-	-	-	
8	-	-	-	-	-	-	-	$E ::= E + T$	

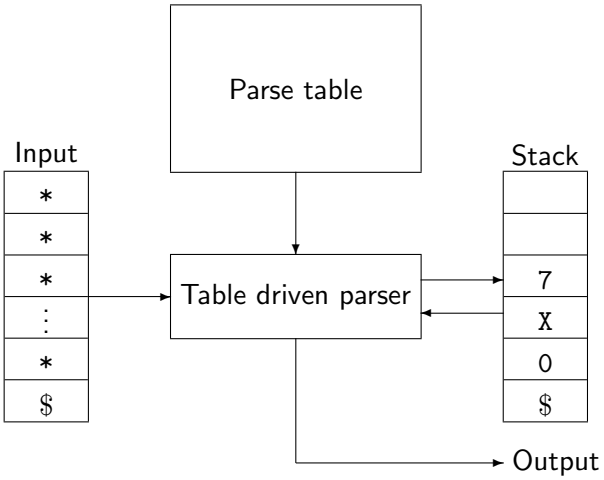
If we use this table to process the string $E + 1;$ we get

```
state      input
  0        E^ + 1 ;
  2        E +^ 1 ;
  7        E + 1^ ;
  5        replace 1 with T, output E + T ;
```

There are LL(1) grammars which are not LR(0), and LR(0) grammars which are not LL(1).

5.4 Stack based implementation

Rather than running the DFA in an LR(0) parser several times feeding its output back into itself, we prefer to set the system up so that we just input the string to be parsed and get as output either success or an error message. We can achieve this by using a stack.



The stack has the EOF character, written here as $, on the bottom and at any time there is a string of states and grammar symbols on the stack, with a state at the top.

We begin with $ and state 0 on the stack. At each stage in the parse, the parser looks at the state on the top of the stack. If it is a non-reducing state then the parser reads the next input symbol, pushes it onto the stack, looks up the state,symbol entry in the parse table and pushes the entry state onto the stack. If there is no entry the parse stops and an error message is given. If the state on the top of the stack is reducing then the appropriate production is found from the table, the symbols and states corresponding to the RHS are popped off the stack, leaving state n, say, on the top. The RHS, A say, of the production and the state at position (n, A) in the table are then pushed onto the stack and the parser proceeds. If the state on the top of the stack is 1 then the input is read and if it is $ (EOF) then the parser stops and reports success.

Example Parse $0 + 1;$.

```
stack                  remaining input
$ 0                       0 + 1 ; $
$ 0 0 4                   + 1 ; $
$ 0 T 3                   + 1 ; $
$ 0 E 2                   + 1 ; $
$ 0 E 2 + 7               1 ; $
$ 0 E 2 + 7 1 5           ; $
$ 0 E 2 + 7 T 8           ; $
$ 0 E 2                   ; $
$ 0 E 2 ; 6               $
$ 0 S 1                   $
return success
```

The string of input symbols which is replaced by a non-terminal when a reduction is carried out (i.e. the string of grammar symbols popped off the stack) is called a *handle*.

5.5 SLR(1) parse tables

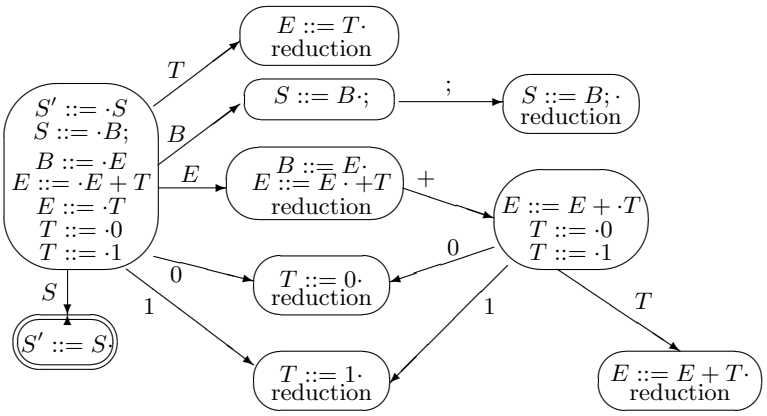
The problem with LR(0) parsing is that a large number of grammars are not LR(0), i.e. their LR(0) parse tables contain some multiple entries and hence the corresponding parser needs to backtrack.

5.5.1 A non-LR(0) grammar

The problem can be easily illustrated by making a slight modification to the LR(0) grammar we used above. The following grammar is obtained by splitting the first production into two parts (it still generates the same language).

```
S' ::= S          S ::= B ;
B ::= E           E ::= E+T | T
T ::= 0 | 1
```

When we construct the LR(0) DFA for this grammar we get



One of the reduction states, 3, contains two production rules. The effect of this is that when we are in this state we don't know whether to reduce, popping E off the stack and pushing B, or to push the next input symbol in the hope that it is +.

We could resolve the problem in this case if we read the next input character before deciding whether to 'shift' or 'reduce'. If the next input is + then we need to push it onto the stack and hope to construct T next. If the next input is ';' then we need to reduce and get B onto the stack. If the input is any other symbol then the parse cannot continue and we report an error.

Reading the next input symbol before deciding whether to shift it onto the stack is known as *one symbol lookahead*. Using one symbol of look ahead vastly increases the class of grammars which can be parsed using the table based techniques we have been considering.

The idea is that reduction states become 'conditional' reduction states; they are reduction states with certain next input symbols, and continuing states with others. If we reduce the stack by replacing a handle with the corresponding non-terminal, A, then the next input symbol needs to be in FOLLOW(A).

We modify the parse table to include extra information which may allow us to decide whether to shift or reduce on a reduction state depending on the

next input character. We construct the states exactly as for the LR(0) parse table. We then put sn (for shift input then state n onto stack) or rk (for reduce using rule k) in entry (m, X) of the parse table, according to the following construction.

5.5.2 Algorithm to construct an SLR(1) parse table

- ◇ Augment the grammar by adding a new special start symbol S' and a production rule $S' ::= S$.
- ◇ Number the productions in the grammar for which the parser is being generated.
- ◇ Construct the start state

$$0 = cl(\{S' ::= \cdot S\})$$

- ◇ For each grammar symbol X, form set of items 0_X and then the closure $cl(0_X)$. If this set is not empty and has not been constructed before it is a new state.
- ◇ For each of the new states constructed and each grammar symbol form the set $cl(n_X)$. Carry on in this way until no new states are constructed.
- ◇ Construct a table which has a row for each state and a column for each grammar symbol and a column for the special EOF symbol \$.
- ◇ For each state n and grammar symbol X, if $m = cl(n_X)$ put sm in the (n, X)th entry of the parse table if X is a terminal and gm if X is a non-terminal.
- ◇ If n contains an item of the form $A ::= \alpha.$, where $A \neq S'$, and if $x \in$ FOLLOW(A) then put rk in the (n, x)th entry of the parse table, where k is the number of the production $A ::= \alpha$.
- ◇ If n contains $S' ::= S\cdot$ put `acc` in the $(n, \$)$ entry of the table.
- ◇ all other table entries are blank (errors).

A *grammar is SLR(1)* if, in the table constructed as above, there is at most one entry in each position.

5.5.3 SLR(1) grammars

Augmenting and numbering the productions for the grammar above we get

0. S'	::=	S	3. E	::=	$E + T$	6. T	::=	1
1. S	::=	B;	4. E	::=	T			
2. B	::=	E	5. T	::=	0			

The SLR(1) states are

$$
\begin{aligned}
0 &= cl(S' ::= \cdot S) = \{S' ::= \cdot S, S ::= \cdot B;, B ::= \cdot E, \\
&\qquad\qquad E ::= \cdot E + T, E ::= \cdot T, T ::= \cdot 0, T ::= \cdot 1\} \\
1 &= cl(O_S) = \{S' ::= S\cdot\} \\
2 &= cl(0_B) = \{S ::= B\cdot;\} \\
3 &= cl(0_E) = \{B ::= E\cdot, E ::= E \cdot + T\} \\
4 &= cl(0_T) = \{E ::= T\cdot\} \\
5 &= cl(0_0) = \{T ::= 0\cdot\} \\
6 &= cl(0_1) = \{T ::= 1\cdot\} \\
7 &= cl(2_;) = \{S ::= B;\cdot\} \\
8 &= cl(3_+) = \{E ::= E + \cdot T, T ::= \cdot 0, T ::= \cdot 1\} \\
9 &= cl(8_T) = \{E ::= E + T\cdot\} \\
5 &= cl(8_0) = \{T ::= 0\cdot\} \\
6 &= cl(8_1) = \{T ::= 1\cdot\}
\end{aligned}
$$

and the follows sets are

$$
\begin{array}{lll}
\textsc{follow}(S') = \{\$\} & \textsc{follow}(S) = \{\$\} & \textsc{follow}(B) = \{;\} \\
\textsc{follow}(E) = \{+\ ;\} & \textsc{follow}(T) = \{+\ ;\} &
\end{array}
$$

and the SLR(1) parse table for this grammar is

	0	1	+	;	$	S	B	E	T
0	s5	s6	-	-	-	g1	g2	g3	g4
1	-	-	-	-	acc	-	-	-	-
2	-	-	-	s7	-	-	-	-	-
3	-	-	s8	r2	-	-	-	-	-
4	-	-	r4	r4	-	-	-	-	-
5	-	-	r5	r5	-	-	-	-	-
6	-	-	r6	r6	-	-	-	-	-
7	-	-	-	-	r1	-	-	-	-
8	s5	s6	-	-	-	-	-	-	g9
9	-	-	r3	r3	-	-	-	-	-

The stack based implementation described for LR(0) parsers requires slight modification to allow the use of SLR parse tables.

5.5.4 Stack based SLR parsing

- ◇ We begin with $ and state 0 on the stack.
- ◇ At each stage in the parse, the parser looks at the current input symbol, x, the state, n, on the top of the stack and looks up the (n, x)th entry in the parse table.
- ◇ If the entry is `acc` the parser terminates and reports success.
- ◇ If the entry is blank the parser terminates and reports an error.
- ◇ If the entry is sm the parser pushes x and then m onto the stack, moves the input pointer on and proceeds to the next stage.

- ◇ If the entry is rk then the parser finds production number k, $A ::= X_1 \dots X_j$ say, and pops the symbols $X_j, \dots, X_1$ and intervening states off the stack. It then looks at the state now at the top of the stack, p say, reads the (p, A)th entry from the parse table, gq say, and pushes A and then q onto the stack and proceeds to the next stage.

Example Parse $0 + 1;$.

```
stack                  remaining input       action
$ 0                       0 + 1 ; $            s5
$ 0 0 5                   + 1 ; $              r5
$ 0 T 4                   + 1 ; $              r4
$ 0 E 3                   + 1 ; $              s8
$ 0 E 3 + 8               1 ; $                s6
$ 0 E 3 + 8 1 6           ; $                  r6
$ 0 E 3 + 8 T 9           ; $                  r3
$ 0 E 3                   ; $                  r2
$ 0 B 2                   ; $                  s7
$ 0 E 2 ; 7               $                    r1
$ 0 S 1                   $                    acc
```

All LR(0) grammars are SLR(1), but there are SLR(1) grammars which are not LR(0). There are LL(1) grammars which are not SLR(1), and SLR(1) grammars which are not LL(1).

5.6 LR(1) and LALR parse tables

There are still many grammars which are not SLR(1). For example, the grammar

$0.S' ::= S$ $\quad$ $1.S ::= L = R$ $\quad$ $2.S ::= R$ $\quad$ $3.L ::= *R$ $\quad$ $4.L ::= a$ $\quad$ $5.R ::= L$

has SLR(1) parse table

	a	$=$	$*$	\$	S	L	R
0	s5	-	s4	-	g1	g2	g3
1	-	-	-	acc	-	-	-
2	-	s6/r5	-	r5	-	-	-
3	-	-	-	r2	-	-	-
4	s5	-	s4	-	-	g8	g7
5	-	r4	-	r4	-	-	-
6	s5	-	s4	-	-	g8	g9
7	-	r3	-	r3	-	-	-
8	-	r5	-	r5	-	-	-
9	-	-	-	r1	-	-	-

which has multiple entries in one place. The problem is that state 2 is $\{S ::= L\cdot = R, R ::= L\cdot\}$ so when we are in this state we don't know whether to reduce the L to R or to carry on hoping to find $= R$.

This can be addressed by using local lookahead instead of the FOLLOW sets, but we shall not discuss this extension in this course. The change is simply a modification to the parse tables, the parsing algorithm itself is the same as for LR(0) and SLR(1) parse tables.

All SLR(1) and LL(1) grammars are LR(1), but there are LR(1) grammars which are not SLR(1) and some which are not LL(1). There are context free grammars for which there is no equivalent LR(1) grammar, but it turns out that in practice LR(1) grammars are general enough to allow us to write real programming languages such as Java and C.

Historically there has been a problem with LR(1) parse tables because in general they can have a very large number of states. It is possible to reduce the number of states down to the same number as for SLR(1) parsers by combining states where the only difference is in the lookahead sets. This reduces the class of grammars which admit parsing using this technique, but it turns out that many real grammars can still be handled. The tables obtained from LR(1) tables by merging states which differ only in their lookahead sets are call LALR tables. These are the tables used by the UNIX standard parser generator, YACC.

5.7 YACC

YACC (Yet Another Compiler-Compiler) is a parser generator based on LALR parsing. The basic structure of the input to YACC is very similar to that of Lex.

declarations
%%
rules
%%
programs

The first %% and the rules section cannot be left out. Each rule is made up of a BNF-style grammar rule and an optional action. The action is executed when the rule is matched. Names representing tokens have to be explicitly declared in the declarations section. The actions are written in C and enclosed within braces. Typically these actions are used to generate a parse tree, directly generate intermediate code or execute actions in an interpreter. YACC can be run with Lex by adding the entry

```
#include "lex.yy.c"
```

to the programs section of the input file.

Figure 12 shows a Lex file called `test.l` and Figure 13 a YACC file called `test.y` that uses the Lex generated lexer specified by `test.l`.

The lex program merely removes white spaces and tabs and recognizes sequences of digits as numbers. The function *atoi* is a C function that converts the string representation of the constant into an integer and returns it to the syntax analyser in the variable `yylval`, and returns the token NUMBER. The

```
%%
[\t] ; /*ignores blanks and tabs*/
[0-9]+ {yylval = atoi(yytext); return NUMBER;}
\n|.   {return yytext[0];}
```

Figure 12 `test.l` – a test lex specification

```
%{
#include<stdio.h>
%}

%token NUMBER

%left '+' '-' /* gives + and - the same precedence */
              /* and makes them left associative */

%%

comm: comm '\n'
|                 /*empty command, no action taken*/
| comm expr '\n'  {printf(``%d\n", $2);}
| comm error '\n' {yyerrok; printf(``Try again \n");}
;

expr: '('expr')' { $$ = $2; }
| expr '+' expr  { $$ = $1 + $3; }
| expr '-' expr  { $$ = $1 - $3; }
| NUMBER
;

%%

#include "lex.yy.c"
yyerror(s)
char*s;
{printf(``%s\n", s);
}
main() {
    return yyparse();
}
```

Figure 13 `test.y` – a YACC test file

operations + and − are not recognized: they are passed directly to the syntax analyser.

The YACC specification describes a program that recognizes sums and differences and converts them to integer equivalents and outputs the result. A basic element of the grammar is a command, represented by the non-terminal `comm`, and a command is a sequence of pairs 'expression, newline'. In this example YACC is employing so-called *syntax directed translation* to build an interpreter: actions, enclosed in braces, appear after each grammar rule which are executed whenever that rule is recognised.

The expression in a command can be empty in which case the action take by the parser is to do nothing. If the expression is valid the action taken is to print the value of the expression and a newline. If the expression is not valid the error value is reset, by *yyerrok*, and an error message is output. The actions associated with the grammar rules for expressions are to calculate the value of the expression. The routine *yyerror* is called whenever the parser detects a syntax error.

To use this example, type these files in, then issue the command

```
lex test.l
```

which creates the file `lex.yy.c` needed by YACC. Then type

```
yacc test.y
```

which creates a C program called `y.tab.c`. Compile this using the command

```
cc y.tab.c -lfl -o calculator
```

and the syntax analyser executable is made and output to the file `calculator`. You can now run it giving it strings of expressions which it will evaluate.

Try typing `calculator <return> 1+1` and the machine should return `2`.

Try typing `3-4-5` and `3-(4-5)` to see the effect of the YACC directive `%left -`.

Now type `((5)` and see what happens.

5.8 Ambiguity in LR parses

A grammar ambiguity will cause a conflict in any type of LR table. We want to be able to use LR parsers with grammars that have the standard 'if-then-else' ambiguity, so we want a longest match strategy.

For an LR parser, to continue to match $\gamma a \beta$ rather than to match γ to A means that we shift the current input symbol a onto the stack rather than perform the reduction $A ::= \gamma$. Thus we implement the longest match strategy by requiring the parser to choose the shift action rather than the reduce action in a shift/reduce conflict. We achieve this by removing the reductions from the shift/reduce conflicts in the parse table.

However, it is important to remember that, except for the case of ambiguity, any strategy for dealing with choices in a parsing technique can cause the parser to incorectly reject some strings. Even in the case of ambiguity, particular derivations will not be found. So it is important only to use such a strategy when you are sure that it's effect will be the one you want.

5.9 Why have a scanner?

We finish the discussion of parsing by briefly considering whether we actually need a lexical analyser. We know that generative grammars are powerful enough to handle the definition of tokens, so why not simply define our context free language grammars over ASCII characters and cut out the lexical analysis stage?

Efficiency : it would be tremendously inefficient to read each single character from a disk file as it is required by the syntax analyser. In the case of a backtracking analyser it might even be impossible on some operating systems because of the difficulty of doing character level seek operations. So the input buffering aspect of scanners is useful.

Even when the entire source text is held in memory, some operations are so common that it is worth coding them specially. For instance, the bulk of many programs is made of alphanumeric identifiers, some of which are keywords. If we implemented keyword lookup by a cascade of `if` statements then significant inefficiencies result. It is much better to recognise that a token begins with an alpha, and then have a special routine that collects alphanumerics and looks up the resulting string in a symbol table. Having a relatively small number of tokens can allow more efficient syntax analysis techniques to be used.

Error reporting : If all of our keywords were implemented at character level within the grammar, then we are 'too close' to the ASCII characters to present useful error messages to the user. When analysing an ill formed string such as

```
if a=b else c=13 then d=14;
```

most real compilers can issue messages along the lines of

```
Line nnnn:  scanned 'else' whilst expecting 'then'
```

When the grammar is defined over tokens it is relatively easy to generate such messages automatically. A definition over characters means that we cannot easily see the wood for trees.

Removing formatting : Early computer languages such as FORTRAN used a fixed format input style with one statement per line, and all statements starting at a fixed point on the line (column 7!).

Most assemblers still impose a fixed format style but nearly all modern languages allow *free format* input in which arbitrary (except sometimes non-zero length) strings containing whitespace characters and/or comments may be inserted between any two language tokens.

To describe such languages purely in a single level context free grammar would require an immense grammar. Instead, the lexical analyser acts as a normalising filter, removing redundant whitespace and comments from the input before the syntax analyser sees it.

6 Syntax analysis III – GLL parsers

General parsers employ parsing techniques which can be used correctly with all grammars and input strings. One of the earliest worst case cubic parsing techniques was published by Jay Earley in 1970, although Earley's description of derivation tree construction was incorrect and giving a correct cubic version turned out to be non-trivial.

There are general parsing techniques which are extensions of recursive descent, and general techniques which are extensions of the LR approach. For example, the LR technique fails if there is a conflict in the LR table. It is possible to extend the technique to grammars whose tables contain conflicts by exploring the execution paths corresponding to each choice. The problem is that unless this is done carefully it can result in an algorithm which is worst case exponential, or even nonterminating if the grammar contains cycles. There do exist general extensions of LR parsers which are worst case cubic.

The main problem with extending the recursive descent technique to general grammars has always been the problem of dealing with left recursion. Of course, it is possible to remove the left recursion by running a preliminary left recursion removal process before building the parser but the parser is then not running on the original grammar, which can made debugging and maintenance difficult. Even if this is done, simple backtracking techniques will still generate exponential parsers in worst case.

In 2009 we published the first version of *Generalised LL* (GLL) parsing, which handles all (including left recursive) context free grammars; runs in worst case cubic time; runs in linear time on LL grammars and which also allows grammar rule factorisation, with consequential speed up. In fact, the construction is so straightforward that implementation by hand is feasible, we constructed a GLL parser for ANSI C before our parser generator tool was built. In this chapter we will give an introduction to the basic, recogniser only, version of GLL. The key to the GLL approach is the explicit handling of what would be the function call stack in a recursive descent parser.

6.1 Introduction to GLL

Recall that a recursive descent parser consists of a collection of parse functions, $parseA()$, one for each non-terminal A in the grammar. The function selects an alternate, α, of the rule for A, according to the current symbol in the input string being parsed, and then calls the parse functions associated with the symbols in α. As we have already seen, it is possible that the current input symbol will not uniquely determine the alternate to be chosen (the grammar is not left factored or is not follow determined), and if A is left recursive the parse function can go into an infinite loop.

GLL parsers extend recursive descent parsers to deal with non-determinism by spawning parallel processes, each with their own stack. This approach is made practical by combining the stacks into a graph structured stack (GSS) which recombines stacks when their associated processes converge. Each of the parallel processes is recorded in a process descriptor, which represents the

process configuration at that point. The descriptors are processed in turn, ensuring that all possible derivation routes are explored. Keeping a record of the descriptors ensures that no exploration path is repeated and this is what makes the parsers worst case cubic.

6.2 Using explicit call stacks

We begin by describing the basic approach using an LL(1) grammar Γ_0. We will then extend the discussion to non-LL(1) grammars.

Consider the grammar Γ_0.

$$\begin{array}{lcl} S & ::= & A\ S\ d \mid B\ S \mid \epsilon \\ A & ::= & a \mid c \\ B & ::= & b \end{array}$$

A traditional recursive descent parser for Γ_0 is composed of parse functions $parseS()$, $parseA()$, $parseB()$ and a main function. We suppose that the input is of length m and is held in a global C-style array I of length $m+1$, where $I[m] = \$$, the end-of-string symbol.

```
main() { i := 0
         parseS()
         if I[i] == $   report success   else  error() }

parseS() {
    if (I[i] == a or I[i] == c}) {
                    parseA()
                    parseS()
                    if (I[i] == d) { i := i + 1 }  else error() }
    else {
      if (I[i] == b) {  parseB()
                        parseS() } } }

parseA() {
     if (I[i] == a) {  if (I[i] == a) { i := i + 1 }  else error() }
     else {
       if  (I[i] == c) {  if  (I[i] == c) { i := i + 1 }  else error() }
       else error() } }

parseB() {
     if  (I[i] == b) {  if  (I[i] == b) { i := i + 1 }  else error() }
     else error() } }
```

We have replaced the $gnt()$ used in traditional RD parsers because a GLL parser will need to access tokens at specified input positions as different process branches are explored. Instead of a variable holding the current input symbol, the global variable i holds the current position of the input pointer, and will be stored as part of the GLL process descriptor in the full version of the approach. The function $error()$ terminates the algorithm and reports failure.

To convert this into a GLL-style parser we turn the function calls into explicit call stack operations using *push*, *pop* and *goto* statements. In a simple algorithm the actions are executed sequentially, in the order that they are written. We can specify a different sequence by labelling actions and using a *goto L* instruction to move the execution to the point labelled L. We label the start of each parse function and the return position from each parse function call. The *goto* statements will jump to the labels at the start of parse functions, the labels of the return positions will be pushed on to a stack s and popped at the end of the function call. Initially the label of the start of the *main* function is put on the stack. In addition, when the stack is popped the label retrieved is put initially into a set $\mathcal{R}$.

Explicit-stack-handling parser for Γ_0

```
     i := 0;  R := ∅;  s := [L0]
     goto LS
L0:  if (R ≠ ∅) remove an element L say from R and  goto L
     else {   if (s == [] and i == m)  return success  else  return failure }
LS:
LS1: if (I[i] == a or I[i] == c})  {
          push(s, R1);  goto LA
R1:       push(s, R2);  goto LS
R2:       if (I[i] == d) { i := i + 1 }  else  goto L0
          pop(s, R)
          goto L0 }
LS2: if (I[i] == b)  {
          push(s, R3);  goto LB
R3:       push(s, R4);  goto LS
R4:       pop(s, R)
          goto L0 }
LS3: if (I[i] == d or I[i] == $})  {
          pop(s, R);  goto L0 }
     goto L0            /* if all tests fail go to L0 */
LA:
LA1: if (I[i] == a) {
          if (I[i] == a) { i := i + 1  }  else  goto L0
          pop(s, R)
          goto L0 }
LA2: if (I[i] == c) {
          if (I[i] == c) { i := i + 1  }  else  goto L0
          pop(s, R)
          goto L0 }
     goto L0
LB:
LB1: if (I[i] == b) {
          if (I[i] == b) { i := i + 1  }  else  goto L0
          pop(s, R)
          goto L0 }
```

goto L_0

Here L_{S_1}, L_{S_2} and L_{S_3} label the blocks of code corresponding to the three alternates in the grammar rule for S, and there is specific code for the ϵ alternate. These labels will be required in the non-LL(1) version. The labels R_1 and R_2 label the return positions of the calls to $parseA()$ and $parseS()$ in the first alternate of S. The other labels have similar roles. There are no calls to an error function. If the parse action fails then the algorithm simply jumps back to the termination code at L_0. Because no element will have been popped from the stack, $\mathcal{R}$ will be empty but s will not, so $failure$ will be returned.

The functions $push()$ and $pop()$ are defined in the natural way as follows.

$push(s, L)\{\ s := [s, L]\ \}$

$pop(s, \mathcal{R})\{$ **if** (s is not empty) {
remove the top element, L say, from s
add L to $\mathcal{R}$ } }

Example We work through this algorithm using the input string *bcd*, keeping track of the values of the variables i, $\mathcal{R}$ and s.

```
I=[b,c,d,$]     m=3     i=0  R={}   s = [L0]

I[0]=b so test at LS1 fails
test at LS2 succeeds
push R3 onto s    i=0   R={}     s=[L0, R3]
goto LB

I[0]=b so test at LB1 succeeds
increment i       i=1   R={}     s=[L0, R3]
pop s             i=1   R={R3}   s=[L0]
goto L0

remove R3 from R  i=1   R={}     s=[L0]
goto L=R3

push R4 onto s    i=1   R={}     s=[L0, R4]
goto LS

I[1]=c so test at LS1  succeeds
push R1 onto s    i=1   R={}     s=[L0, R4, R1]
goto LA
test at LA1 fails
test at LA2 succeeds
increment i       i=2   R={}     s=[L0, R4, R1]
pop s             i=2   R={R1}   s=[L0, R4]
goto L0
```

```
remove R1 from R  i=2   R={}     s=[L0, R4]
goto R1

push R2 onto s    i=2   R={}     s=[L0, R4, R2]
goto LS
I[2]=d so test at LS1 fails
test at LS2 fails
test at LS3 succeeds
pop s             i=2   R={R2}   s=[L0, R4]
goto L0

remove R2 from R  i=2   R={}     s=[L0, R4]
goto R2
increment i       i=3   R={}     s=[L0, R4]
pop s             i=3   R={R4}   s=[L0]
goto L0

remove R4 from R  i=3   R={}     s=[]
goto R4
pop s             i=3   R={L0}   s=[]
goto L0

remove L0 from R  i=3   R={}     s=[]
goto L0
R is empty, s=[], i=3=m  so return success
```

Additional Exercises Work through the above algorithm in the same way using the strings (i) *acd*, (ii) *b*, (iii) *aadd*, (iv) ϵ, (v) *da*, (vi) *add*, and (vii) *ccd*, recording the values of the variables i, $\mathcal{R}$ and s at each step.

6.3 Non-LL(1) grammars - using elementary descriptors

If we have a grammar which is not LL(1) then simply directly handling the call stack is not sufficient. There can be several alternates of a nonterminal with the same element in their FIRST sets and we need to execute each choice. To do this efficiently we use descriptors to record the current parser configuration at the point of a choice, and store the descriptors in a set $\mathcal{R}$ for subsequent processing. The outer structure of a GLL algorithm is thus a loop which removes an element from $\mathcal{R}$ and continues the parse from the configuration recorded in that element. The algorithm terminates when there are no further descriptors to be processed.

It is possible for the parser to get into the same configuration in different ways and to avoid processing the same descriptor more than once we maintain a set $\mathcal{U}$ of all descriptors that have been added to $\mathcal{R}$.

The parser configuration at any point consists of the line of the algorithm the parser is at, the call stack and the current input position. So we initially use elementary descriptors of the form (L, s, i), where s is a complete stack.

This will be inadequate in general because there may be infinitely many stacks, which is why we refer to these as elementary descriptors. In the full version of the algorithm we will combine the stacks into a single structure and just record the node corresponding to the top of the stack. First, however, we illustrate the way in which descriptors are used with just elementary descriptors and the grammar Γ_1 below.

$$\begin{array}{lcl} S & ::= & A\ S\ d \mid B\ S \mid \epsilon \\ A & ::= & a \mid c \\ B & ::= & a\ d \mid b \end{array}$$

A traditional recursive descent parser for Γ_1 has the form

```
main() { i := 0
         parseS()
         if I[i] == $   report success   else error() }

parseS() {
    if (I[i] == a or I[i] == c}) {
                    parseA()
                    parseS()
                    if (I[i] == d) { i := i + 1 }  else error() }
    else {
      if (I[i] == a or I[i] == b}) {  parseB()
                                      parseS() } } }

parseA() {
     if (I[i] == a) {  if (I[i] == a) { i := i + 1 }  else error() }
     else {
       if  (I[i] == c) {  if  (I[i] == c) { i := i + 1 }  else error() }
       else error() } }

parseB() {
     if  (I[i] == a) {  if  (I[i] == a) { i := i + 1 }  else error()
                        if  (I[i] == d) { i := i + 1 }  else error() }
     else {
       if  (I[i] == b) {  if  (I[i] == b) { i := i + 1 }  else error() }
       else error() } }
```

Γ_1 is not LL(1) so, as we know, this algorithm will not behave correctly without some additional mechanism for dealing with non-determinism.

An *elementary descriptor* is a triple (L, s, j) where L is a line label, s is a stack and j is a position in the input array I. We maintain a set $\mathcal{R}$ of current descriptors. When a particular execution thread of the algorithm stops, at input $I[i]$ say, the top element L is popped from the stack $s = [s', L]$ and (L, s', i) is added to $\mathcal{R}$ (if it has not already been added). We use $pop(s, i, \mathcal{R})$ to denote this action. Then the next descriptor (L', t, h) is removed from $\mathcal{R}$ and execution starts at line L' with stack t and input symbol position h. The overall execution terminates when the set $\mathcal{R}$ is empty.

In order to allow us, later, to combine stacks we record both the line label L and the current input buffer index k, using the notation (L, k). We use a function $push(s, L, k)$ which simply updates the stack s by pushing on the element (L, k).

Each time a nonterminal is encountered, where the RD parser makes a parse function call, a GLL algorithm applies a test against the first set of each alternate and, for each alternate which passes the test, an elementary descriptor is created and stored for subsequent processing. Also, where an RD parse function would have terminated, a GLL parser creates an elementary descriptor, via the pop function, and this stored for processing.

Below is an elementary GLL algorithm for Γ_1, assuming that the input is of length m and is held in an array I.

Elementary-descriptor-based parser for Γ_1

$i := 0$; $s := [(L_0, 0)]$; $\mathcal{U} := \{(L_s, s, i)\}$; $\mathcal{R} := \emptyset$
goto L_S

L_0: **if** $(\mathcal{R} \neq \emptyset)$ { remove (L, s_1, j) from $\mathcal{R}$
$s := s_1$; $i := j$; **goto** L }
else {
if $((L_0, [\,], m) \in \mathcal{U})$ report success **else** report failure }

L_S: **if** $(I[i] \in \{a, c\})$ $add((L_{S_1}, s, i), \mathcal{R})$
if $(I[i] \in \{a, b\})$ $add((L_{S_2}, s, i), \mathcal{R})$
if $(I[i] \in \{d, \$\})$ $add((L_{S_3}, s, i), \mathcal{R})$
goto L_0

L_{S_1}: $push(s, R_1, i)$; **goto** L_A
R_1: $push(s, R_2, i)$; **goto** L_S
R_2: **if** $(I[i] == d)$ { $i := i + 1$ } **else** **goto** L_0
$pop(s, i, \mathcal{R})$; **goto** L_0

L_{S_2}: $push(s, R_3, i)$; **goto** L_B
R_3: $push(s, R_4, i)$; **goto** L_S
R_4: $pop(s, i, \mathcal{R})$; **goto** L_0

L_{S_3}: $pop(s, i, \mathcal{R})$; **goto** L_0

L_A: **if** $(I[i] \in \{a\})$ $add((L_{A_1}, s, i), \mathcal{R})$
if $(I[i] \in \{c\})$ $add((L_{A_2}, s, i), \mathcal{R})$
goto L_0

L_{A_1}: **if** $(I[i] == a)$ { $i := i + 1$ } **else** **goto** L_0
$pop(s, i, \mathcal{R})$; **goto** L_0

L_{A_2}: **if** $(I[i] == c)$ { $i := i + 1$ } **else** **goto** L_0
$pop(s, i, \mathcal{R})$; **goto** L_0

L_B: **if** $(I[i] \in \{a\})$ $add((L_{B_1}, s, i), \mathcal{R})$
if $(I[i] \in \{b\})$ $add((L_{B_2}, s, i), \mathcal{R})$
goto L_0

L_{B_1}: **if** $(I[i] == a)$ $\{\ i := i + 1\ \}$ **else** **goto** L_0
if $(I[i] == d)$ $\{\ i := i + 1\ \}$ **else** **goto** L_0
$pop(s, i, \mathcal{R})$; **goto** L_0

L_{B_2}: **if** $(I[i] == b)$ $\{\ i := i + 1\ \}$ **else** **goto** L_0
$pop(s, i, \mathcal{R})$; **goto** L_0

Below are the definitions of the support functions that are being used.

$push(s, L, i)\{\ s := [s, (L, i)]\ \}$

$pop(s, i, \mathcal{R})\{$ **if** (s is not empty) {
remove the top element, (L, j) say, from s to get s'
if ((L, s', i) is not in $\mathcal{U}$) {
add (L, s', i) to $\mathcal{R}$ and to $\mathcal{U}$ } } }

$add((L, s, i), \mathcal{R})\{$ **if** ((L, s, i) is not in $\mathcal{U}$) {
add (L, s, i) to $\mathcal{R}$ and to $\mathcal{U}$ } }

Example As an example we execute the above algorithm with input ad, keeping track of the values of the variables i, $\mathcal{R}$ and s. In this example we will only report the values of variables when they change.

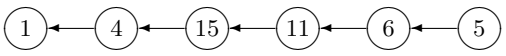
```
I=[a,d,$]      m=2
i=0   R={}   s = [(L0,0)]   D1 = (L0,[(L0,0)],0)   U = {D1}

goto LS   I[0]=a so two descriptors are created at LS
    D2 = (LS1,[(L0,0)],0)  D3 = (LS2,[(L0,0)],0)   R={ D2, D3 }
goto L0   remove D2 from R   s=[(L0,0)]  i=0  R={ D3 }
goto LS1  push(s,R1,0)     s=[(L0,0) (R1,0)]
goto LA   I[0]=a so one descriptor is added
    D4 = (LA1,[(L0,0) (R1,0)],0)         R={ D3, D4 }
goto L0   remove D3 from R     s=[(L0,0)]  i=0  R={ D4 }
goto LS2  push(s,R3,0)     s=[(L0,0) (R3,0)]
goto LB   I[0]=a so one descriptor is added
    D5 = (LB1,[(L0,0) (R3,0)],0)     R={ D4, D5 }
goto L0   remove D4 from R     s=[(L0,0) (R1,0)]  i=0  R={ D5 }
goto LA1  increment i,   i=1
          pop s, so one new descriptor is added
    D6 = (R1, [(L0,0)], 1)     R={ D5, D6 }
goto L0   remove D5 from R     s=[(L0,0) (R3,0)]  i=0  R={ D6 }
goto LB1  I[0]=a so increment i,    i=1
          I[1]=d so increment i,    i=2
```

```
          pop s, so one new descriptor is added
    D7 = (R3, [(L0,0)], 2)       R={ D6, D7 }
goto L0   remove D6 from R     s=[(L0,0)]  i=1  R={ D7 }
goto R1   push(s,R2,1)     s=[(L0,0) (R2,1)]
goto LS   I[1]=d so one descriptor is added
    D8 = (LS3,[(L0,0) (R2,1)],1)      R={ D7, D8 }
goto L0   remove D7 from R     s=[(L0,0)]  i=2  R={ D8 }
goto R3   push(s,R4,2)     s=[(L0,0) (R4,2)]
goto LS   I[2]=$ so one descriptor is added
    D9 = (LS3,[(L0,0) (R4,2)],2)      R={ D8, D9 }
goto L0   remove D8 from R     s=[(L0,0) (R2,1)]  i=1  R={ D9 }
goto LS3  pop s, so one new descriptor is added
    D10 = (R2, [(L0,0)], 1)      R={ D9, D10 }
goto L0   remove D9 from R     s=[(L0,0) (R4,2)]  i=2  R={ D10 }
goto LS3  pop s, so one new descriptor is added
    D11 = (R4, [(L0,0)], 2)      R={ D10, D11 }
goto L0   remove D10 from R     s=[(L0,0)]  i=1  R={ D11 }
goto R2   increment i,   i=2
          pop s, so one new descriptor is added
    D12 = (L0, [], 2)       R={ D11, D12 }
goto L0   remove D11 from R     s=[(L0,0)]  i=2  R={ D12 }
goto R4   pop s, the descriptor (L0, [], 2) is in U,  so no action
    U={ D1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12 }
goto L0   remove D12 from R     s=[]  i=2  R={ }
goto L0
R is empty, (L0, [], 2) is in U, report success
```

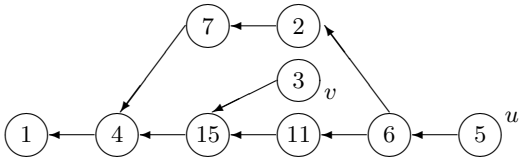
6.4 The GSS and the set $\mathcal{P}$

The problem with the approach as it is described above is that for some grammars the number of descriptors created can be exponential in the size of input, and the process does not always terminate for grammars with left recursion. We deal with these issues by combining the stacks into a single, global graph structure, a graph structured stack (GSS), recording only the corresponding stack top node in the descriptor, and using loops in the GSS when left recursion is encountered.

We can represent any stack as a chain of nodes labelled with the stack elements. For example $[1, 4, 15, 11, 6, 5]$ can be represented as



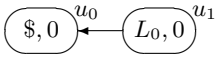
If we had, say, two other stacks, $[1, 4, 7, 2, 6, 5]$ and $[1, 4, 15, 3]$, then these could be combined with the first into a graph structure.



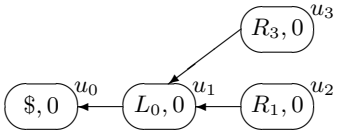
The node u represents both the first two stacks and the node v represents the third stack. Space is saved by sharing common parts of the stacks, and, for GLL parsers, the number of descriptors is reduced by combining elementary descriptors whose stacks have a common top element.

To understand the basic concept of a GSS as used in our GLL parsers, consider again the grammar Γ_1 and input ab.

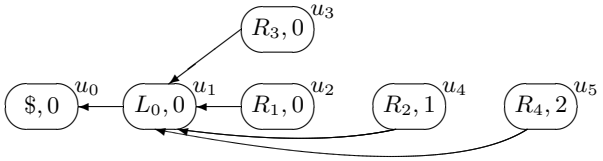
We can represent the initial stack $[(L_0, 0)]$ as a single node u_1 labelled $(L_0, 0)$. Because we shall record nodes rather than full stacks we need a way to represent the empty stack. Thus we create a dummy base node, u_0, labelled $(\$, 0)$ and make this node a child of u_1. Then u_0 represents the empty stack and u_1 represents the stack $[(L_0, 0)]$.



The stacks $[(L_0, 0)(R_1, 0)]$ and $[(L_0, 0)(R_3, 0)]$ can be combined with the first stack, and are represented by u_2 and u_3 respectively.



We can carry on in this way and build a GSS which contains all the stacks used in the above example.



Then, for example, the elementary descriptor $D8$ above is written as full descriptor in the form $D8 = (LS3, u_4, 1)$.

The grammar Γ_1 does not illustrate the real power of a GSS because each GSS node represents only one stack, so the total number of descriptors is not reduced. We shall consider a further example, but first we describe the modified push and pop stack operations that are needed to build a GSS.

A *descriptor* is a triple (L, s, i) where L is a label, s is a GSS node and i is an integer. Because a GSS node can represent the top of many stacks, the action in $pop(s, i, \mathcal{R})$ is replaced by actions which add (L_s, v, i) to $\mathcal{R}$ for all children v of s, where L_s is the line label in s.

A problem arises in the case when an additional child, w say, is added to a node u after a pop statement has been executed, because the pop action also needs to be applied to this child. To address this we use a set $\mathcal{P}$ which contains pairs (u, k) for which a 'pop' has been executed. When a new child node w is added to u, for all $(u, k) \in \mathcal{P}$ if $(L_u, w, k) \notin U$, where L_u is the line label in u, then (L_u, w, k) is added to $\mathcal{R}$ and $\mathcal{U}$. This contingent pop application is carried out by a modification to the push function. Because this modification is extensive we call the function $create()$ rather than $push()$.

The function $create(L, u, j)$ creates a GSS node $v = (L, j)$ with child u, if one does not already exist, and returns v. If $(v, k) \in \mathcal{P}$ then $add((L, u, k), \mathcal{R})$ is called.

$add((L, u, j), \mathcal{R})$ { **if** $((L, u, j) \notin U$ { add (L, u, j) to U and to $\mathcal{R}$ } }

$pop(u, j, \mathcal{R})$ { **if** $(u \neq u_0)$ { add (u, j) to $\mathcal{P}$
for each child v of u { $add((L_u, v, j), \mathcal{R})$ } } }

$create(L, u, j)$ { **if** there is not already a GSS node labelled (L, j) create one
let v be the GSS node labelled (L, j)
if there is not an edge from v to u {
create an edge from v to u
for all $((v, k) \in \mathcal{P})$ { $add((L, u, k), \mathcal{R})$ } }
return v }

6.5 Example - a GLL parser

We consider the grammar Γ_2

$$\begin{array}{lcl} S & ::= & a\ B\ c \mid A\ B\ d \\ A & ::= & a \mid c \\ B & ::= & b\ B \mid \epsilon \end{array}$$

A GLL parser for Γ_2

create GSS nodes $u_1 = (L_0, 0)$, $u_0 = (\$, 0)$ and an edge from u_1 to u_0
$i := 0$; $s := u_1$; $\mathcal{U} := \{(L_S, u_1, 0)\}$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$
goto L_S

L_0: **if** $(\mathcal{R} \neq \emptyset)$ { remove (L, s_1, j) from $\mathcal{R}$
$s := s_1$; $i := j$; **goto** L }
else { **if** $((L_0, u_0, m) \in \mathcal{U})$ report success **else** report failure }

L_S: **if** $(I[i] \in \{a\})$ $add((L_{S_1}, s, i), \mathcal{R})$
if $(I[i] \in \{a, c\})$ $add((L_{S_2}, s, i), \mathcal{R})$
goto L_0

L_{S_1}: **if** $(I[i] == a)$ { $i := i + 1$ } **else** **goto** L_0
$s := create(R_1, s, i)$; **goto** L_B
R_1: **if** $(I[i] == c)$ { $i := i + 1$ } **else** **goto** L_0
$pop(s, i, \mathcal{R})$; **goto** L_0

L_{S_2}: $s := create(R_2, s, i)$; **goto** L_A
R_2: $s := create(R_3, s, i)$; **goto** L_B
R_3: **if** $(I[i] == d)$ { $i := i + 1$ } **else** **goto** L_0
$pop(s, i, \mathcal{R})$; **goto** L_0

L_A: **if** $(I[i] \in \{a\})$ $add((L_{A_1}, s, i), \mathcal{R})$

if $(I[i] \in \{c\})$ $add((L_{A_2}, s, i), \mathcal{R})$
goto L_0

L_{A_1}: **if** $(I[i] == a)$ { $i := i + 1$ } **else** **goto** L_0
$pop(s, i, \mathcal{R})$; **goto** L_0

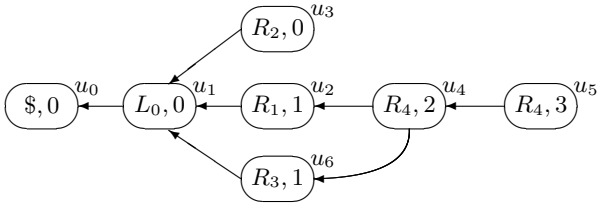
L_{A_2}: **if** $(I[i] == c)$ { $i := i + 1$ } **else** **goto** L_0
$pop(s, i, \mathcal{R})$; **goto** L_0

L_B: **if** $(I[i] \in \{b\})$ $add((L_{B_1}, s, i), \mathcal{R})$
if $(I[i] \in \{c, d\})$ $add((L_{B_2}, s, i), \mathcal{R})$
goto L_0

L_{B_1}: **if** $(I[i] == b)$ { $i := i + 1$ } **else** **goto** L_0
$s := create(R_4, s, i)$; **goto** L_B
R_4: $pop(s, i, \mathcal{R})$; **goto** L_0

L_{B_2}: $pop(s, i, \mathcal{R})$; **goto** L_0

We execute this algorithm for Γ_2 with input *abbc*. The final GSS will be




```
I=[a,b,b,c,$]  m=4  i=0   R={}  s=u1   D1=(L0,u1,0)   U = {D1}

goto LS    I[0]=a so two descriptors are created
    D2 = (LS1,u1,0)  D3 = (LS2,u1,0)
goto L0    remove D2 from R   s=u1  i=0
goto LS1   I[0]=a so increment i,    i=1
           create(R1,u1,1)     s=u2
goto LB    I[1]=b so one descriptor is added
    D4 = (LB1,u2,1)
goto L0    remove D3 from R     s=u1  i=0
goto LS2   create(R2,u1,0)     s=u3
goto LA    I[0]=a so one descriptor is added
    D5 = (LA1,u3,0)
goto L0    remove D4 from R     s=u2  i=1
goto LB1   I[1]=b so increment i,    i=2
           create(R4,u2,2)     s=u4
goto LB    I[1]=b so one descriptor is added
    D6 = (LB1,u4,2)
goto L0    remove D5 from R     s=u3  i=0
```

```
goto LA1   I[0]=a so increment i,    i=1
           pop u3, so one new descriptor is added
    D7 = (R2,u1,1)  P={(u3,1)}
goto L0    remove D6 from R     s=u4  i=2
goto LB1   I[0]=b so increment i,    i=3
           create(R4,u4,3)     s=u5
goto LB    I[3]=c so one descriptor is added
    D8 = (LB2,u5,3)
goto L0    remove D7 from R     s=u1  i=1
goto R2    create(R3,u1,1)     s=u6
goto LBI   [1]=b so one descriptor is added
    D9 = (LB1,u6,1)
goto L0    remove D8 from R     s=u5  i=3
goto LB2   pop u5, so one new descriptor is added
    D10 = (R4,u4,3)    P={(u3,1), (u5,3)}
goto L0    remove D9 from R     s=u6  i=1
goto LB1   I[1]=b so increment i,    i=2
           create(u6,R4,2)     s=u4
    /* a node (R4,2) already exists so a new child u6 is simply added
       to u4, and two stacks recombine */
goto LB    I[2]=b,
    /* the descriptor (LB1,u4,2) is already in U so no action */
goto L0    remove D10 from R     s=u4  i=3
goto R4    pop u4, this node has two children so two descriptors are added
    D11 = (R4,u2,3)  D12 = (R4,u6,3)   P={(u3,1), (u5,3), (u4,3)}
goto L0    remove D11 from R     s=u2  i=3
goto R4    pop u2, so one new descriptor is added
    D13 = (R1,u1,3)   P={(u3,1), (u5,3), (u4,3), (u2,3)}
goto L0    remove D12 from R     s=u6  i=3
goto R4    pop u6, so one new descriptor is added
    D14 = (R3,u1,3)    P={(u3,1), (u5,3), (u4,3), (u2,3), (u6,3)}
goto L0    remove D13 from R     s=u1  i=3
goto R1    I[3]=c so increment i,    i=4
           pop u1, so one new descriptor is added
    D15 = (L0,u0,4)    P={(u3,1), (u5,3), (u4,3), (u2,3), (u6,3), (u1,4)}
goto L0    remove D14 from R     s=u1  i=3
goto R3    I[3]=c so no action
goto L0    remove D15 from R     s=u0  i=4
goto L0    R is empty, (L0,u0,4) is in U, report success
```

6.6 Formal templates for generating GLL parsers

Like a recursive descent parser, a GLL parser follows the grammar closely, and can be generated by writing a line of code for each grammar symbol instance. In this section we give templates for writing these lines of code.

We present simple versions of the templates. However, it is easy to make a GLL parser more efficient by extending the templates to remove some tests,

to add some tests before calls to *pop*() and *create*(), and to run in recursive descent mode when a nonterminal is LL(1). To keep the templates simple we do not do any of these in this course.

We also note that it is possible, although not trivial, to extend the GLL approach so that the algorithms produce derivation trees, and so that it can be applied directly to EBNF grammars. The main issue that needs to be addressed for derivation tree generation is that for ambiguous grammars there will be more than one derivation tree, so an efficient way of representing all the trees is required. For EBNF the tricky part is dealing with `[x]` and `{x}` in the cases where $x \stackrel{*}{\Rightarrow} \epsilon$.

Finally we note that when a nonterminal, A, is encountered in a GLL parser we need to test each of its alternates, α, against the current input symbol, a. Essentially we need to check that that a is in $\textsc{first}(\alpha)$, but this is not quite enough when α derives ϵ. Thus we define the set $\textsc{select}(\alpha, A)$, which is used in the templates.

$$\textsc{select}(\alpha, A) = \begin{cases} (\textsc{first}(\alpha) \backslash \{\epsilon\}) \cup \textsc{follow}(A) & \text{if } \epsilon \in \textsc{first}(\alpha) \\ \textsc{first}(\alpha) & \text{if } \epsilon \notin \textsc{first}(\alpha) \end{cases}$$

We now give the templates for generating a GLL parser (recogniser) from a BNF grammar.

Each nonterminal instance on the right hand sides of the grammar rules is given an instance number. We write A_k to indicate the kth instance of nonterminal A. Each alternate of the grammar rule for a nonterminal is also given an instance number. We write $A ::= \alpha_k$ to indicate the kth alternate of the grammar rule for A.

The templates use the following globally defined identifiers:

m is a constant integer whose value is the length of the input
I is a constant integer array of size $m + 1$ containing the input
i is an integer
GSS is a digraph whose nodes are labelled with elements of the form (L, j)
s is a GSS node
$\mathcal{P}$ is a set of GSS node and integer pairs
$\mathcal{R}$ is a set of descriptors

For a terminal a we define

$$code(a) \;=\; \textbf{if}(I[i] == a)\ \{\ i := i + 1\ \}\ \ \textbf{else}\ \{\ \textbf{goto}\ L_0\ \}$$

For a nonterminal instance A_k, where $B ::= \alpha A \beta$ say, we define

$$\begin{array}{rcl} code(A_k) & = & s := create(R_{A_k}, s, i);\ \textbf{goto}\ L_A \\ & & R_{A_k}: \end{array}$$

For each production $A ::= \alpha_k$ we define $code(A ::= \alpha_k)$ as follows. Let $\alpha_k = x_1 x_2 \ldots x_f$, where each x_p, $1 \leq p \leq f$, is either a terminal or a nonterminal

instance of the form X_l. We also allow $f = 0$ so we can have $\alpha_k = \epsilon$. Then we define

$$\begin{array}{lll} code(A ::= \alpha_k) & = & code(x_1) \\ & & code(x_2) \\ & & \ldots \\ & & code(x_f) \\ & & pop(s, i, \mathcal{R}); \ \ \textbf{goto}\ L_0 \end{array}$$

For the grammar rule $A ::= \alpha_1 \mid \ldots \mid \alpha_t$. we define $code(A)$ as follows.

$$\begin{array}{lll} code(A) & = & \textbf{if}(I[i] \in \textsc{select}(\alpha_1, A))\ add((L_{A_1}, s, i), \mathcal{R}) \\ & & \ldots \\ & & \textbf{if}(I[i] \in \textsc{select}(\alpha_t, A))\ add((L_{A_t}, s, i), \mathcal{R}) \\ & & \textbf{goto}\ L_0 \\ & & L_{A_1} : code(A ::= \alpha_1) \\ & & \ldots \\ & & L_{A_t} : code(A ::= \alpha_t) \end{array}$$

Then, supposing that the nonterminals of the grammar Γ are $A, \ldots, X$, the GLL recognition algorithm for Γ has the form:

```
    read the input into I and set I[m] := $
    create GSS nodes u_1 = (L_0, 0), u_0 = ($, 0) and an edge from u_1 to u_0
    i = 0; s := u_1; R := ∅; U := {(L_S, u_1, 0)}; P := ∅
    goto L_S
L_0: if R ≠ ∅ {
        remove a descriptor, (L, u, j) say, from R
        s := u, i := j,  goto L }
    else  if ((L_0, u_0, m) ∈ U) { report success }  else { report failure }

L_A: code(A)
   ...
L_X: code(X)
```

7 Semantic evaluation

We now start to consider the synthesis of the object code. We need to generate code corresponding to the intended meaning of the source program. The writer of a language must also provide the semantics. This is usually done with reference to the grammar of the language. The semantic analyser then uses the information provided to construct code, often in an intermediate language. Because the semantics are usually defined with reference to the grammar, the semantic analyzer is designed to build on the work of the parser.

The techniques for semantic analysis are not as well developed as those for parsing. There is no general algorithmic method for defining the semantics of a language. (In fact this is usually done by describing properties using the English language.) We will look briefly at the two techniques, syntax directed translation and top down translation, using as examples simple semantics that can be specified in a precise way, e.g. basic mathematical operations and types of things such as real or integer numbers.

7.1 Tokens and attributes

The lexical analyser returns a stream of tokens to the parser, but these tokens originally had more attached information. For tokens whose pattern contains more than one string there is the lexeme which was found when the token was matched. This lexeme is not lost; it is usually stored in the symbol table and it can be retrieved by the semantic analyser.

We shall write `<token,` *lexeme*`>` to show that the token `token` was returned because the lexeme *lexeme* was scanned. For example, `< ID,` *fred1*`>` is the token `ID` which was returned when the identifier `fred1` was scanned. We shall also use `token`.*lexeme* to refer to the lexeme attached to the particular instance of that token. So the statements '`ID` such that `ID`.*lexeme=fred1*' is equivalent to the statement `<ID,` *fred1*`>`.

Tokens can have other properties. For example, if the token is an identifier or a number it may have an associated type. It may also have a value (which is different in different parts of the code).

We call properties of grammar symbols *attributes*. An attribute can be any property; for example a value, a lexeme, or a pointer to a place in a symbol table.

We attach attributes to non-terminals as well as tokens. The idea is to attach the attributes in such a way that the parse tree can be 'walked', evaluating the attributes at each node, so that the output can be fed into the back end of the compiler. For example, the output can be intermediate code (see next section) or it may be the actual result of 'running' the input program.

7.2 Annotated parse trees

We assign a set of semantic rules to each production rule of the grammar. The purpose of the semantic rules is to tell the semantic analyser how to evaluate

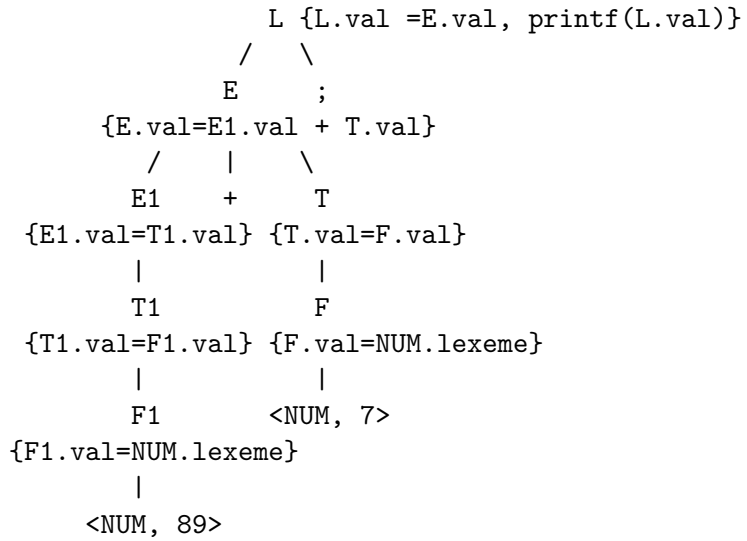
the attribute of a node in the parse tree from those attributes it has already calculated.

For example, the following grammar generates arithmetic expressions. `NUM` is a token whose pattern is the decimal numbers. The corresponding semantic rules define the value of the left hand side of a rule in terms of the values of its right hand side.

```
Grammar Rules                Semantic Rules

L ::= E ;                    { L.val = E.val; printf(L.val)}
E ::= E1 + T                 { E.val = E1.val + T.val}
E ::= T                      { E.val = T.val}
T ::= T1 * F                 { T.val = T1.val * F.val}
T ::= F                      { T.val = F.val}
F ::= NUM                    { F.val = NUM.lexeme}
```

The attribute `NUM`.lexeme is the value of the particular token `NUM` which has been passed on by the lexical analyser. Each of the elements T.val, E.val etc, are attributes for T, E, etc.

We have had to be careful here. Where there were two occurrences of the same grammar symbol in a production we have had to give them distinguishing names so that we can tell when we apply the rule which expression is supposed to denote the attribute required. For example, the `E1.val` on the RHS of `E.val := E1.val + T.val` is the `E.val` arising from the actual instance of the RHS of the production used, the `E.val` on the left is the one that arises from the actual instance of E on the LHS.



If the attributes are evaluated in turn, the value of the root node will be the value of the input expression. So in this case we can use the parse tree to actually execute the input program.

A parse tree that shows the attributes at each node is called an *annotated parse tree*.

7.3 Syntax directed translation

In a *syntax directed translator* the parser constructs an annotated parse tree and then the semantic analyzer walks the tree, evaluating attributes at each node.

We cannot evaluate a node until we have evaluated the attributes needed. For example, each of the attributes `F.val`, `T.val`, etc., in the above example, is defined in terms of the previous one. The rules effectively say 'make the attribute for T whatever this attribute for F is' or 'make `T.val` the product of `S.val` and `F.val`' etc. Thus we cannot evaluate `T.val` until we have evaluated the appropriate `F.val`.

The methods that we can use to walk an annotated tree are determined by properties of the semantic rules. There are two common types of attribute, **synthesized attributes** and **inherited attributes**. An attribute is inherited it is determined by looking only at the attributes of its parent and siblings in the tree.

If all of the attributes are synthesized, an attribute can be determined by looking at the attributes of its children in the parse tree and we can perform semantic analysis by starting with the attributes of the programming construct being considered, i.e. the string of tokens that are the leaves of the complete parse tree, and working up the parse tree constructing attributes as we go, using the rules associated with the relevant productions. Finally we end at the top of the tree with the semantic value of the whole input string.

7.4 Attribute grammars

Formally, an **attribute grammar** is a grammar in which each symbol has an associated set (possibly empty) of synthesized and inherited attributes and each production $A ::= \gamma$ has an associated set of semantic rules of the form $b = f(c_1, \ldots, c_k)$ where either

1. b is a synthesized attribute of A and $c_1, \ldots, c_k$ are attributes of A and the symbols that make up γ, or

2. b is an inherited attribute of a symbol in γ and $c_1, \ldots, c_k$ are attributes of A and the symbols that make up γ.

Example
We can use inherited attributes to express the dependence of a token on the context in which it appears. For example, consider a language which has integers and reals. Suppose also that the programming language allows us to declare various identifiers to be real or integer. This could be done with a programming construct of the form `real x1, ... , xn` whose intended meaning is that the identifiers `x1, ... , xn` have real values.

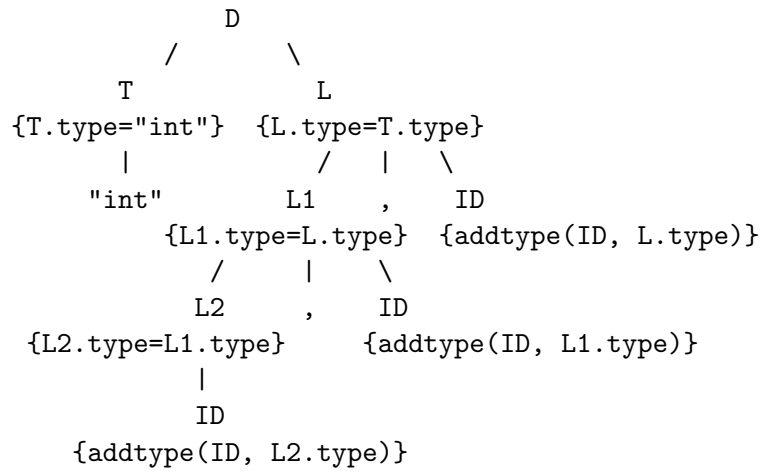
```
Grammar Rules              Semantic Rules

D ::= TL.                  {L.type = T.type}
T ::= "int".               {T.type = "int"}
T ::= "real".              {T.type = "real"}
L ::= L1 , ID.             {L1.type = L.type, addtype(ID, L.type)}
L ::= ID.                  {addtype(ID, L.type)}
```

For the 4th production there are two semantic rules, the first is to make L.type for the L on the right equal to the L.type for the L on the left, the second is an instruction that calls a procedure *addtype*, whose function is to add the information in its second argument to the entry in the symbol table in which the type of the first argument is stored.

The non-terminal D has no attributes, L has one which is inherited, T has one attribute which is synthesized and we can view the rule addtype() as defining a dummy inherited attribute of `ID`. The tokens 'integer' and 'real' are viewed as being constant properties of attributes, or nullary functions. (Strictly speaking this is a syntax-directed definition, it updates the symbol table as a side effect.)

Certain types of inherited attributes can be evaluated on the way down on a depth first walk of the parse tree. Synthesized attributes can be evaluated on the walk back up during such a tree walk.

For example, the following annotated parse tree can be produced from the string `"int" ID, ID, ID`



An attribute grammar is called **S-attribute** if it only has synthesized attributes. The semantic analysis of S-attribute grammars is easy to carry out, one just begins with the left-most available branch of the tree and works upwards.

An attribute grammar is called **L-attribute** if, for each production $A ::= x_1 \ldots x_n$, an inherited attribute of x_i depends only on the attributes of $x_1, \ldots, x_{i-1}$ and the inherited attributes of A.

If we have an L-attribute grammar, a syntax directed translator can walk an annotated parse tree using a top down depth first walk, evaluating the inherited attributes on the way down and the synthesized attributes on the way back up.

Furthermore, if we have an L-attribute grammar it is possible to evaluate the attributes while the parse tree is being constructed, if the parsing technique is top down, depth first.

7.5 Top down translation

We now consider top-down parsing with concurrent semantic evaluation.

A **translation scheme** is a context-free grammar in which attributes are associated with grammar symbols and semantic actions have been inserted in the RHS of productions. The actions are enclosed within braces. We can think of a translation scheme as a grammar by thinking of the actions as special tokens.

A *top down translator* has a grammar and a corresponding translation scheme. It parses input strings using a top down left-most depth first search, evaluating the semantic rules in the translation scheme as the parse is carried out.

Example

Suppose the source language is the set of arithmetic expressions that are sums and products of integers grouped together using parentheses. We define a token `INT` whose pattern is the set of strings of digits.

Suppose that the object language is the polish notation for arithmetic expressions. When a string is input the compiler has to check that it is a string in the source language and produce a string in the object language with the same meaning, in this case the equivalent expression in polish notation. We add actions to the grammar to get a translation scheme that determines the semantics. The actions in this case can be written so that they produce the output code.

```
Translation Scheme

E ::= E+T {printf(+)} | T.
T ::= T*F {printf(*)} | F.
F ::= (E) | NUM {printf(NUM.lexeme)}.
```

Here `NUM`.*lexeme* is the lexeme of the particular instance of the token `NUM`, which has been stored in the attribute table by the lexical analyser.

On input of the string (7+8)∗3 the lexical analyser produces the string `(<NUM,7>+<NUM,8>)*<NUM,3>`. If the parser is top down depth-first left-most it then produces, using the translation scheme, the derivation:

$$\begin{aligned} E \Rightarrow T &\Rightarrow T * F\{printf(*)\} \\ &\overset{*}{\Rightarrow} (E) * F\{printf(*)\} \\ &\overset{*}{\Rightarrow} (NUM\{printf(NUM.lexeme)\} + NUM\{printf(NUM.lexeme)\} \end{aligned}$$

$$\{printf(+)\}) * NUM\{printf(NUM.lexeme)\}\{printf(*)\}.$$

The 'intermediate code' is

```
printf(7)
printf(8)
printf(+)
printf(3)
printf(*)
```

A simple translator can then be run to read this output and carry out the print actions. The final output of the compiler is then 7 8 + 3 ∗.

Since actions can be procedure calls it is possible to generate a wide variety of types of intermediate code using translation schemes.

7.6 Types

It is common practice to 'overload' operations. Strictly speaking an operation comes with a specified range and domain. However, it is usual to use the same notation for two operations with different ranges and domains if the operations are somehow 'the same'. For example addition of integers takes a pair of integers and returns another integer, while addition of matrices takes two matrices and returns another. We usually denote both these operations by the symbol +.

We can use typing to deal with overloaded operations, we provide rules that say, for example, if n and m are integers then the '+' in $n + m$ is addition of integers and the result is to have type integer, etc. But we cannot have n + M, where M is a 3×3 matrix! The general idea is to assign types to grammar symbols, some symbols have predetermined types, the types of others are inferred from **type constructors**, which are just rules for inferring types.

The rules for inferring types can be written as translation schemes. In section 7.4 is an example of a translation scheme which allows us to infer the types of a list of numbers from a declaration.

Suppose that we have a language in which we can write integer and real identifiers, and pointers to integers, and a function *val* that returns the value of the identifier pointed to by the pointer. So *val* has as its domain the set of pointers to integers and as its range the set of integers. Thus it only makes sense to apply *val* to identifiers of type 'pointer' but this can't easily be specified syntactically. We use attributes to assign the appropriate types and check at the semantic analysis stage that *val* is used correctly. We use the following translation scheme.

```
E ::= ID       {E.type = lookuptype(ID)}
E ::= val(E1)  {E.type = if (E1.type=="intPtr") return "int"
                         else type_error()}
E ::= E1 + E2  {E.type =
                 if(E1.type=="int" & E2.type=="int") return "int"
                 else
```

```
if (E1.type =="real" & E2.type=="real")
    return "real"
else type_error()}
```

During the lexical analysis tokens `ID`, `val`, `+`, `(` and `)` are created. Using a function `type_error()` that runs a suitable error reporting procedure and a function `lookuptype()` that finds the type of its argument by consulting the symbol table, we can write a translation scheme that checks that *val* and '+' are being used correctly, i.e. applied to the right sorts of objects.

The compiler accepts (makes no comment on) a statement of the form `ID + val(ID)` if first `ID` has type "int" and the second `ID` has type "intPtr", but it reports a type_error if it meets the statement `ID + ID` if one `ID` is of type "int" and the other is of type "intPtr", or if it meets the statement `val(ID)` where `ID` has type "int".

7.7 Semantic actions in rdp

In this section we consider the design of interpreters for a tiny language called `mini`. The language includes variable declaration, assignment, the four basic arithmetic operators and a `print` procedure that can output a mix of variables and strings, much like the Pascal `writeln` statement.

7.7.1 Adding interpreter semantics

The grammar of Figure 11 from Chapter 4 can be used to produce a syntax checker for `mini` that parses `mini` code but does not actually execute a `mini` program. In this section, the grammar will be *decorated* with attributes and semantic actions that describe a usable interpreter for `mini`. The fully annotated grammar is shown in Figure 14.

The two `mini` grammars differ in only three respects. The

1. the decorated grammar has a `SYMBOL_TABLE` declaration,
2. attributes (denoted by a colon and an identifier) have been appended to some names and
3. C-language fragments delimited by `[* ... *]` brackets describing the interpreter semantics have been inserted in some productions.

7.7.2 Symbol table manipulation

Symbol table manipulation is fundamental to the operation of most language translators. `rdp` provides a `SYMBOL_TABLE` directive that provides an interface to the `symbol` module in the support library. The directive creates a named table with a specified size and hash key. You must also specify compare, hash and print functions and the user data to be stored in each record. The symbol table module in `rdp_supp` provides routines such as

```
TITLE("Mini V1.50 (c) Adrian Johnstone 1997")
SUFFIX("m")

SYMBOL_TABLE(mini 101 31
             symbol_compare_string
             symbol_hash_string
             symbol_print_string
             [* char* id; integer i; *]
            )

program   ::= {[var_dec | statement ] ';' }.

var_dec   ::= 'int' ( ID:name [ '=' e1:val ]
            [* mini_cast(symbol_insert_key
            (mini, &name, sizeof(char*), sizeof(mini_data)))->i = val; *]
               )@','.

statement ::= ID:name
              [* if (symbol_lookup_key(mini, &name, NULL) == NULL) {
                   text_message(TEXT_ERROR, "Undeclared variable '%s'\n", name);
                   symbol_insert_key(mini, &name, sizeof(char*), sizeof(mini_data));
                 }
              *]
              '=' e1:val
              [* mini_cast(symbol_lookup_key(mini, &name, NULL))->i = val; *] |

              'print' '(' ( e1:val [* printf("%li", val); *] |
                            String:str [* printf("%s", str); *]
                          )@','
                    ')'.

e1:integer ::= e2:result {'+' e2:right [* result += right; *] |
                         '-' e2:right [* result -= right; *] }.

e2:integer ::= e3:result {'*' e3:right [* result *= right; *] |
                         '/' e3:right [* result /= right; *] }.

e3:integer ::= '+' e4:result |
               '-' e4:result [* result = -result; *] |
               e4:result.

e4:integer ::= ID:name
               [* if (symbol_lookup_key(mini, &name, NULL) == NULL) {
                    text_message(TEXT_ERROR, "Undeclared variable '%s'\n", name);
                    symbol_insert_key(mini, &name, sizeof(char*), sizeof(mini_data));
                  }
               *]
               [* result = mini_cast(symbol_lookup_key(mini, &name, NULL))->i; *] |
               INTEGER:result |
               '(' e1:result ')'.

comment    ::= COMMENT_NEST('(*' '*)').

String: char * ::= STRING_ESC('"' '\\'):result.
```

Figure 14 The `mini` grammar decorated with interpreter semantics

- ◇ `symbol_lookup_key()` which returns a pointer to the symbol table record for an identifier and
- ◇ `symbol_insert_key()` which creates a new symbol table record containing an identifier string with space for user data.

These routines both return `void` pointers, and so `rdp` automatically creates a macro called *name*`_cast()` (where `name` is the symbol table name) which casts a void pointer to a pointer to the user data type.

7.7.3 Attributes

Each `rdp` parser production can return zero or one synthesized attributes which can be any C-language datatype including a `struct`. By using `structs`, multiple values may be returned by a production. A production that does not return an attribute is called a void production. `rdp` also provides *inherited attributes* which may be passed down into productions like parameters. A extension of `mini` uses inherited attributes to implement an IF ... THEN ... ELSE statement.

In the `mini` grammar, all productions are void except for `e1, e2, e3, e4` and `string`. These productions are used to evaluate simple expressions and to parse strings for use in the `print` statement.

Most built-in primitives also return a value. The `ID` primitive returns the identifier's name as a character string and the `INTEGER` primitive returns the numeric value of the integer parsed.

Within the body of a production, synthesized attributes are named as return values (specified by a colon and an attribute name after the production call). Local variables are automatically declared in the equivalent C parser function to receive the attribute values. In addition, in a non-void production a variable called `result` is automatically declared to receive the return value. The same restrictions apply to attribute names as exist for production names: they must not contain two consecutive underscores and must not conflict with C reserved words or library function names.

7.7.4 Semantic actions

Semantic actions may be inserted anywhere in the production and they may make use of attributes or explicitly declared variables. The expression analyser productions illustrate this best: the production for multiply and divide is

```
e2:integer ::= e3:result {'*' e3:right [* result *= right; *] |
                          '/' e3:right [* result /= right; *] }.
```

As the expression is parsed, the left hand operand is evaluated and stored in variable `result`, the multiply or divide operator is parsed and then the right hand operand similarly evaluated into the attribute variable `right`. The semantic actions then update the `result` variable by multiplying (or dividing) in the right hand operand.

```
Recursive descent parser generator V1.50 (c) Adrian Johnstone 1997
Generated on Jan 09 1998 13:12:35 and compiled on Jan  9 1998 at 12:58:16

******: Checking for continuation tokens
******: No continuation tokens needed
******: Checking for empty alternates
******: Warning - rule 'comment' never called so deleted
******: 9 rules, 14 tokens, 0 actions, 36 subrules
******: Generating first sets
******: Generating follow sets
******: Follow sets stabilised after 9 passes
******: Checking for clashes with reserved words
******: Checking for disjoint first sets
******: Checking for nested nullable subrules
******: Checking nullable rules
******: Updating follow sets
******: Follow sets stabilised after 3 passes
******: Dumping header file to 'mini.h'
******: Dumping parser file to 'mini.c'
******: Text buffer size 34000 bytes with 29640 bytes free
******: 0 errors and 1 warning
******: 0.069 CPU seconds used
```

Figure 15 Output of `rdp -v -omini minicalc`

7.7.5 Generating and running the interpreter

A copy of the `mini` grammar is supplied in the standard distribution. The C language parser source file is generated with the following command

```
rdp -v -omini minicalc
```

The `-v` flag sets *verbose* mode which causes `rdp` to generate statistics and informational messages. A log of such an `rdp` run is shown in Figure 15.

The warning message about production `comment` is expected. This production is a dummy used only to set up the `COMMENT` delimiters. Since `comment` is never called from anywhere else in the grammar it need not be instantiated into the C parser source file, indeed if it were it would probably generate a warning message from the C compiler. `rdp` automatically deletes productions that are never called.

The `mini` interpreter evaluates expressions and executes `print` statements. Figure 16 shows the result of running `mini` on `testcalc.m` with the scanner listing switched on with a `-l` option.

```
Mini V1.50 (c) Adrian Johnstone 1997
Generated on Jan 09 1998 18:23:30 and compiled on Jan  9 1998 at 18:23:23

******:
     1: (*****************************************************************************
     2: *
     3: * RDP release 1.50 by Adrian Johnstone (A.Johnstone@rhbnc.ac.uk) 16 August 1997
     4: *
     5: * test.m - a piece of Mini source to test the Mini interpreter
     6: *
     7: * This file may be freely distributed. Please mail improvements to the author.
     8: *
     9: *****************************************************************************)
    10:
    11: int a=3+4, b=1;
    12:
a is 7
    13: print("a is ", a, "\n");
    14:
    15: b=a*2;
    16:
b is 14, -b is -14
    17: print("b is ", b, ", -b is ", -b, "\n");
    18:
    19: int z = a;
7
    20: print(z, "\n");
    21:
    22: (* End of test.m *)
    23:
******: 0 errors and 0 warnings
******: 0.153978 CPU seconds used
```

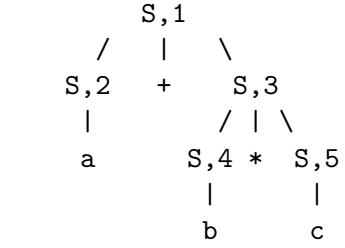
Figure 16 Output of `mini -l -v testcalc.m`

8 Intermediate Code

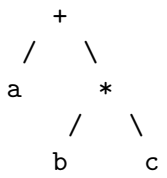
Intermediate code is usually written in a low level language whose constructs are thought of as being basic operations on a hypothetical machine. The idea is that the code is independent of the particular target machine but is sufficiently like machine code that the translation into the target code is simple and direct. In this section we use semantic rules to generate three address code, a form of intermediate code. But first we consider a more compact form of parse tree.

8.1 Abstract parse trees

Most derivation trees contain more information than we need.

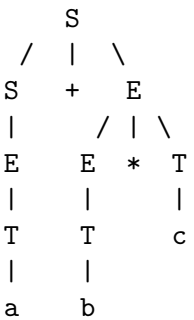


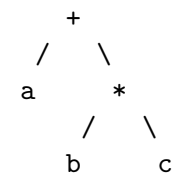
We only need the structure of the input, not the way in which that structure was established. The tree below has enough information to evaluate the input.



The value of the node labelled `*` is the value of `b` times the value of `c`. The value of the node labelled `+`, and hence the value of the whole input, is the value of `a` plus the value of the node labelled `*`.

We construct a reduced version of a derivation tree starting at the root and working down. If a node has more than one child then label the node with appropriate tokens (usually some of those which appear in the right hand side of the associated production rule) and remove the children which are token nodes from the tree. If the node has only one child, merge the parent and child node, labelling the merged node with the label of the child.

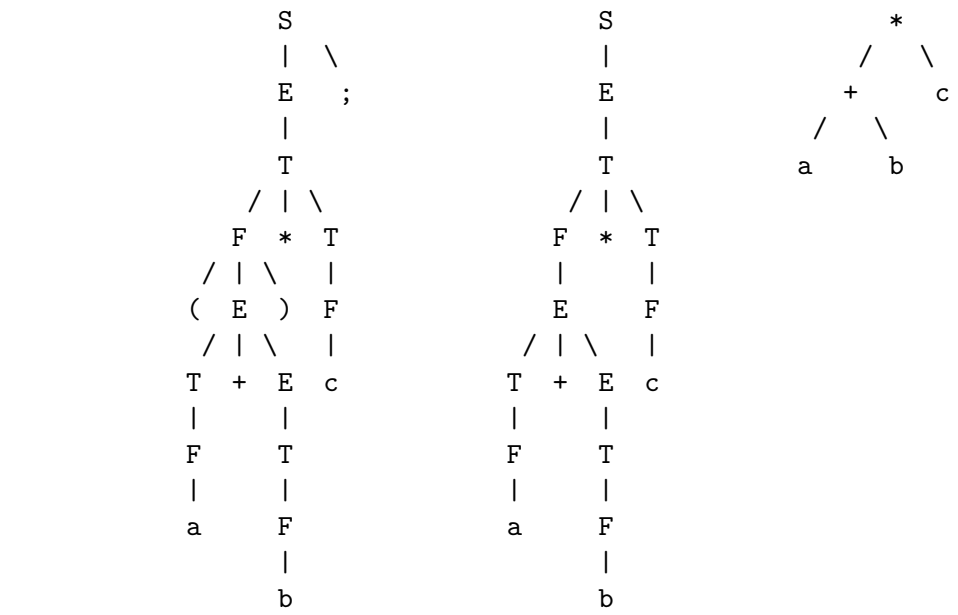




A tree constructed in this way is called an *abstract parse tree.*

Some of the tokens may be just 'syntactic sugar' in the sense that once the derivation tree is constructed they are no longer needed.

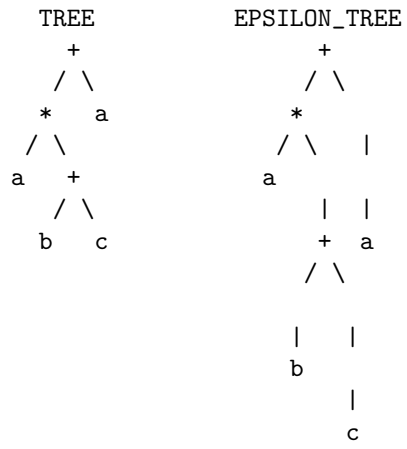
```
S ::= E ;
E ::= T+E | T
T ::= F*T | F
F ::= (E) | a | b | c

(a+b)*c
```



The grammar writer can tell in advance which tokens are not necessary for semantic evaluation. In `rdp` it is possible to specify a 'promotion' operator which suppresses the building of such nodes by the parser. A single operator `^` causes the node for the corresponding symbol to be placed 'under' it's parent node in the derivation tree. A double operator `^^` causes the node for the corresponding symbol to be placed 'over' it's parent node in the derivation tree.

```
S ::= E^^ ;^ .
E ::= T [ +^^ E ]:^^ .
T ::= F [ *^^ T ]:^^ .
F ::= (^ E^^ )^ | a^^ | b^^ | c^^ .
```

In `rdp` we specify ϵ using [] brackets. To attach promotion operators to ϵ we use a colon at the end of the brackets. So `[..]:^^` means place the node for ϵ over its parent node. The tree produced by the above grammar on input $a * (b + c) + a;$ is shown on the right below. `rdp` runs a post parse routine that removes the ϵ nodes to get the tree on the left below.



To see the tree without the ϵ nodes removed use the `EPSILON_TREE` directive in `rdp` instead of the `TREE` directive.

It is often possible to 'evaluate' the input from its abstract parse tree by creating simple code. Each interior node is assigned a temporary variable which carries the value of the subtree below it. A relatively simple assembler can then be written to evaluate the assignment expressions.

```
      * t1          t2 := a + b
    /   \           t1 := t2 * c
t2 +     c
 /   \
a     b
```

(It is traditional to use the Pascal style assignment symbol in the statements produced.)

What we are actually doing here is evaluating attributes according to fairly obvious semantic rules. This idea is the basis of a standard method of generating a form of intermediate code called three address code.

8.2 Three address code

Three address code is a sequence of statements of the form

```
x := y op z,
```

where `x,y,z` are names either from the symbol table or temporary names generated by the compiler that behave like names from the symbol table, and `op` is an operator that takes two arguments. We allow some of the elements to be missing, for example the statements may be of the form

```
x := y,  or  x := - z,
```

but we cannot have statements like

```
x := y + z * w  or   y + z * w.
```

The name three address code comes from the fact that each basic operation in the language usually requires three addresses, two for the operands and one for the result.

The statements are viewed as instructions for a hypothetical machine that has a set of addresses and hard-wired operations corresponding to each operation in the set of statements. The machine is expected to carry out the instructions in the order that it reads them.

`x := y op z` is viewed as being the instruction 'perform the operation `op` on the contents of addresses y and z and store the result at address x'.

`x:=y` is thought of as assigning the value of y to x, i.e. copy the value at address y to address x.

`x:= op y` is thought of as performing the unary operation `op`, e.g. negation, conversion to floating point number, etc. on y and storing the result in x.

In addition to these operation instructions a three address code will also have some 'control of flow statements' such as

`goto L` and `if t goto L`.

`goto L` – carry out the instruction labeled L next and then continue from there working through the instructions.

`if t goto L` – t is a boolean expression such as `x < y`, which evaluates to true or false. If the value is `true` move to the instruction labeled L and carry on, otherwise proceed to the next instruction.

There may be more complicated assignment statements like

`x := y[i]`,

corresponding to the instruction to read into x the value in the address that is i beyond the address y.

For a given source language the choice of statements allowed in the three address code is an important part of the compiler design. The more types of statement the more flexible the code and the easier the writing of the translator, but the more difficult to generate the final target code. The above examples are typical of the types of statement normally used, but should not be seen as the only possibilities.

8.2.1 Generating three address code

We generate three address code by ascribing semantic rules to productions. We need several different types of attribute. One attribute `x.place` will be a symbolic identifier representing the address at which the value of `x` is stored. If x is a token of type `ID` or `num` then we shall take the lexeme of `x` to be `x.place`. If `x` is a non-terminal then we shall generate a new temporary name to be `x.place`.

If `x` is the token `ID` or `num` we assume that `x.place` has been passed on by the lexical analyser. Since `x.place` is a new name when `x` is a non-terminal

there must also be a procedure that returns new temporary names as they are needed. We shall assume that the procedure *newtemp* does this.

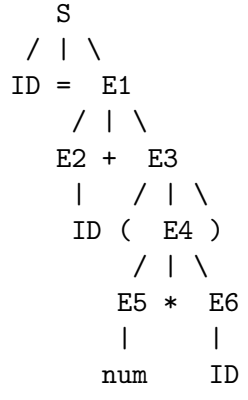
The attributes `x.code` are intended to be the portion of three address code so far produced. At the end of the translation `S.code`, where `S` is the start symbol, will contain the complete three address version of the code.

Suppose that our source language allows constructions of the form `a = e`, where a is an identifier, e is an arithmetic expression involving sums and products of numbers and identifiers and whose intended meaning is that a should contain the result of evaluating the expression e. The grammar contains the following productions:

```
S ::= ID = E      S.code = E.code
                           ID.place ‘:=’ E.place

E ::= E1 + E2     E.place = newtemp
                  E.code = E1.code
                           E2.code
                           E.place ‘:=’ E1.place ‘+’ E2.place

E ::= E1 * E2     E.place = newtemp
                  E.code = E1.code
                           E2.code
                           E.place ‘:=’ E1.place ‘*’ E2.place

E ::= (E1)        E.place = E1.place
                  E.code = E1.code

E ::= ID          E.place = ID.place
                  E.code =‘ ’

E ::= num         E.place = num.place
                  E.code = ‘ ’
```

When a construct S of this sort has been parsed the corresponding intermediate three address code will be contained in the attribute `S.code`.
Example:

```
price = tax + (3 * cost)
   ID = ID + (num * ID)
```



```
E6.code := ' '                          S.code :=  t1 := 3 * cost
E5.code := ' '                                     t2 := tax + t1
E4.code := t1 := 3 * cost                          price := t2
E3.code := t1 := 3 * cost
E2.code := ‘ ’
E1.code := t1 := 3 * cost
           t2 := tax + t1
```

(Here ‘ ’ passes nothing.)

8.2.2 Flow of control statements

We need to have more machinery to cope with flow-of-control statements. For example, an instruction of the form ‘if B then S’ has the intended meaning that if B is true then the procedure S should be carried out, otherwise proceed to the next instruction. We specify the syntax of such statements by including the production `S ::= if B then S` in the grammar. If B is a simple Boolean expression of the form x *relop* y, where *relop* is a relational operator such as $<$ then such statements can be implemented in three address code using labels. We assume that we have three address code, `B.code`, that places the value 1 in the address `B.place` if B holds and places 0 there otherwise. We also assume that we have already generated the three address code, `S1.code`, corresponding to the S on the RHS of the production, that there are two label attributes, `S.true` and `S.false`, associated with S, and that there is some procedure *newlabel* which returns a new label name each time it is called.

The semantic rules associated with the production `S ::= if B then S1` are:

```
S.code =           B.code
                   'if' B.place 'goto' S.true
                   'goto' S.false
          S.true : S1.code
          S.false:
```

Here `S.false` labels the empty statement. When this statement is encountered the program execution simply moves on to the next step.

Sequences of statements and sequences surrounded by Begin/End statements are integrated directly into three address intermediate code, because the flow of control is specified directly.

```
S ::= S1 , S2           S.code = S1.code
                                 S2.code
S ::= begin S1 end      S.code = S1.code
```

An instruction of the form ‘while B do S’ has the intended meaning that while B holds the procedure S should be carried out. What semantic rules should be associated with the production `S ::= while B do S`? We generate the three address code, `S1.code`, corresponding to the S on the RHS of the

production. We need two label attributes, `S.begin` and `S.after`, to mark the beginning and end of the while loop S.

Then the semantic rules associated with the production `S ::= while B do S1` are:

```
S.begin = newlabel
S.after = newlabel
S.false = newtemp
S.code  = S.begin : B.code
                    S.false := B.place '=' 0
                    'if' S.false 'goto' S.after
                    S1.code
                    'goto' S.begin
          S.after :
```

We can extend the three address code operators to include an `ifn` operator. We define `ifn t goto L` as: if `t` is false move to the instruction labeled L and carry on, otherwise proceed to the next instruction. This allows an alternative attribute rule for a while statement which does not require a new temporary variable.

```
S.begin = newlabel
S.after = newlabel
S.code  = S.begin : B.code
                    'ifn' B.place 'goto' S.after
                    S1.code
                    'goto' S.begin
          S.after :
```

8.2.3 Example 1

The following 'program' adds the numbers 1×2, 2×3, ... , (n–1)×n together and calls the result `sum`:

```
x = 1  , z = 0 ,
while x < n do
    begin
    z = z+x*(x+1),  x = x + 1
    end ,
sum = z
```

We assume that the lexical analyser produces:

```
ID = num , ID = num ,  while ID < num do
begin ID = ID + ID * (ID + num),  ID = ID + num  end , ID = ID
```

We also suppose that the source language grammar contains the productions:

```
S ::= S , S | begin S end | ID = E | while B do S
B ::= ID < E
```

```
E ::= T+E | T
T ::= F*T | F
F ::= ID | num
```

The first and the last three production rules are a subset of the productions discussed above, and we have seen how to assign semantic rules to get three address code from these. We will use the `if` version of the while semantics. To determine semantic rules for the production `B ::= ID < E` the code `B.code` generated from B must assign `true` to `B.place` if B is true and `false` to `B.place` otherwise.

```
B.place =  newtemp
B.code  =  E.code
           B.place ':=' ID.place '<' E.place
```

The semantic rules for the production `E ::= T` are

```
E.place =  T.place
E.code  =  T.code
```

Using the rules to calculate the three address code for

```
           z = z + x * (x+1)  and for   x = x + 1   gives

S1.code =  t4 := x + 1
           t3 := x * t4
           t2 := z + t3
            z := t2
           t5 := x + 1
            x := t5
```

Then the full three address code version of the source program is:

```
     x := 1
     z := 0
L1 : t1 := x < n
     t6 := t1=0
     if t6 goto L2
     t4 := x + 1
     t3 := x * t4
     t2 := z + t3
     z := t2
     t5 := x + 1
     x := t5
     goto L1
L2 :
     sum := z
```

8.2.4 Arrays in three address code

We assume that the entries in the array are stored in consecutive addresses. We shall denote by *base* the first address of section allocated to the array and we shall assume that the width of the array elements is w, so each entry in the array is w addresses long, the first address is assumed to contain the value of the element. We take the elements of the array and write them in a list.

To write the code we need to know the way in which the list has been formed from A. If A is a $1 \times n$ or $n \times 1$ array, i.e. a list, then we use the natural order, A : $a_1, a_2, \ldots, a_n$. For an $n \times m$ array there are two standard ways, row-major and column-major.

```
Row-major : A[1,1]    Column-major : A[1,1]
            A[1,2]                   A[2,1]
              .                        .
            A[1,m]                   A[n,1]
              .                        .
            A[n,m]                   A[n,m]
```

For a 1-dimensional array the ith entry A[i] begins at the address *base + (i–1)×w*. Note that *base – w = c* is a constant. The code

```
t1 := i * w
 c := base - w
t2 := c[t1]
```

stores the value `A[i]` at the address `t2`.

For an $n \times m$ array A stored in row-major form, the entry A[i,j] begins at address *base + ((i–1)×m + j – 1)×w*. Then c = *base –(m + 1)×w* is a constant and the code

```
t1 := i * w
t2 := t1 * m
t3 := j * w
t4 := t2 + t3
t5 := m + 1
t6 := t5 * w
 c := base - t6
t7 := c[t4]
```

stores the value `A[i,j]` at the address `t7`.

8.2.5 Example 2

Suppose that A and B are real vectors of length 10, so they are 1×10 dimensional arrays. Suppose also that each entry in the array has two attributes, its value and its type. So $w = 2$. The following three address code forms the scalar product of A and B and stores the result at the address `prod`.

```
      prod := 0
         i := 1
L :    t1 := i * 2
       c_A := base_A - 2
        t2 := c_A[t1]
        t3 := i * 2
       c_B := base_B - 2
        t4 := c_B[t3]
        t5 := t2 * t4
        t6 := prod + t5
      prod := t6
        t7 := i+1
         i := t7
        t8 := i < 11
      if t8 goto L
```

We can see by inspection that the codes in the two examples are not very efficient! We shall consider this issue in Section 9.

9 Code improvement

It is not possible to write a routine that will always generate the most optimal code for any input. The main aim of the optimization phase is to try to improve loops and procedure calls. All changes must be safe, i.e. must definitely not change the semantics of the program, so they tend to be conservative. We shall concentrate on optimization of three address code.

9.1 Basic blocks

What we do is to divide the code into self contained sections called basic blocks. Basic blocks will be defined so that the end result of executing the code in a block is to assign values to names. The idea is that we can change the code within a basic block any way that we like so long as the end result is still that the same values are assigned to the same names. Such optimization will be safe. The goto statements by which we get into and out of blocks can be thought of as joining the blocks together, creating the flow of the program.

Let P be a sequence of three address statements. A statement in P is a *leader* if it is either:

1. the first statement of P,
2. the statement labelled L (or the statement after if this is empty) where there is a statement 'goto L' or 'if x relop y goto L',
3. the next statement following a (conditional or unconditional) goto statement in P.

The *basic block* of a leader in the three address code P is the sequence of statements which begins with the leader and includes everything up to (but excluding) the next leader.

For Example 1 in Section 8:

```
LEADERS:
x := 1              by (1)                    x := 1
t1 := x<n           by (2)                    z := 0
sum := z            by (2) or (3)          -----------------
t4 := x + 1         by (3)                 L1: t1 := x < n
                                               t6 := t1 = 0
                                               if t6 goto L2
                                           -----------------
                                               t4 := x + 1
                                               t3 := x * t4
                                               t2 := z + t3
                                               z := t2
                                               t5 := x + 1
                                               x := t5
                                               goto L1
                                           -----------------
                                           L2: sum := z
```

A name x in a basic block is said to be *live* in the block if its value is used at a later stage in the program. Otherwise we say that x is *dead* in the block.

Two basic blocks are *equivalent* if they compute the same values for the same live names.

The following operations, which can be performed on basic blocks, are called *structure preserving transformations*. They transform a block into an equivalent block, i.e. performing them does not change the meaning of the program.

Common subexpression elimination

If two statements compute *the same value* then one can be replaced with an assignment of the LHS to the LHS of the remaining statement. If an expression has already been computed and the values of its variables have not changed since the computation then we can use the computed expression rather than recomputing it.

We can modify the third block in the above example to get

```
t4 := x + 1
t3 := x * t4
t2 := z + t3
 z := t2
t5 := t4
 x := t5
goto L1
```

We need to be careful with subexpression elimination. Consider the code

```
price := tax + e
  val := cost + e
 cost := tax + e
  tax := cost + e
```

We can eliminate `cost := tax + e` and replace it with `cost := price`,

```
price := tax + e
  val := cost + e
 cost := price
  tax := cost + e
```

but we cannot replace the LHS of `val` or `tax` because whatever `cost` was when `val` was defined it went up by `e` before `tax` was defined so `val` and `tax` do not compute the same value.

Dead-code elimination

If x is dead in the block then statements of the form `x := ...` can be deleted.

Renaming temporary variables

If t is a temporary name introduced by the compiler we can replace t by a new temporary u provided we replace all the occurrences of t in the program by u.

Interchange of statements

If we have two consecutive statements in the block such that the LHS of each does not appear on the LHS or the RHS of the other, then the order of the two statements can be swapped. In Example 1 of Section 7 we can swap the order of the statements `z := t2` and `t5 := x+1`.

Algebraic simplification

We can replace `x := y + 0` or `x := y * 1` by `x := y`, and `x := y * 0` by `x := 0` (this is sometimes called *constant folding*). It may be 'cheaper' to replace `x := y ↑ 2` with `x := y * y` (sometimes called *strength reduction*).

Copy propagation

Given a statement `x := y`, where x is dead outside the basic block, we can replace all occurrences of x after this statement by y. The point of this is that x then becomes dead at the point of the assignment `x := y` and thus this statement can be deleted using rule 2 above.

Examples

Using this to transform the code from Example 2 in Section 8 gives:

```
      prod := 0
         i := 1
L :    t1 := i * 2
       c_A := base_A - 2
        t2 := c_A[t1]
       c_B := base_B - 2
        t4 := c_B[t1]
        t5 := t2 * t4
        t6 := prod + t5
      prod := t6
        t7 := i+1
         i := t7
        t8 := i < 11
      if t8 goto L
```

Transforming the code from Example 1 gives:

```
      x := 1
      z := 0
L1 :  t1 := x < n
      t6 := t1=0
      if t6 goto L2
      t4 := x + 1
      t3 := x * t4
      t2 := z + t3
```

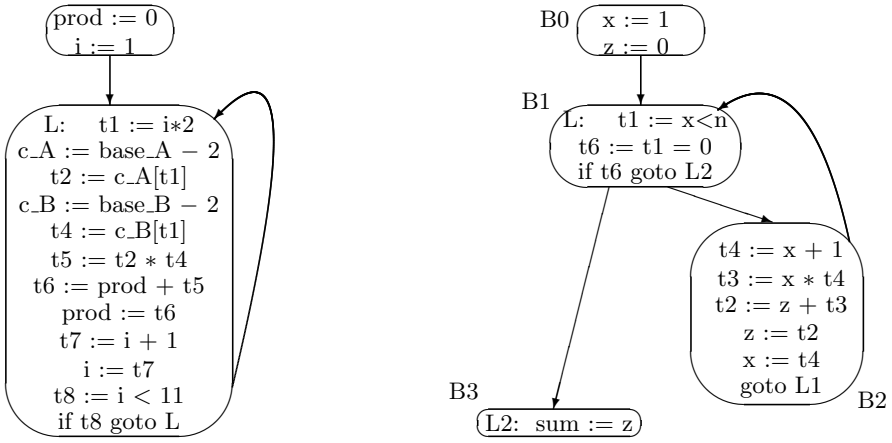
```
        z := t2
        x := t4
       goto L1
L2 :
       sum := z
```

9.2 Flow Graphs

So far we have looked at optimizing code within a block. We now consider optimizing the numbers and order of the blocks themselves.

A *flow graph* is a directed graph whose nodes are the basic blocks of some three address program. There is a distinguished first node: that node whose leader is the first statement of the code. This is called the *initial* node. There is a directed edge from the block B1 to the block B2 if there is a conditional or unconditional goto statement at the end of B1 which goes to the first statement of B2, or if B1 does not end in an unconditional goto and the first statement of B2 follows the last statement of B1 in the program sequence.

The flow graphs for the partially optimized versions of Examples 1 and 2:



Let B and B′ be nodes of a flow graph. We say that B *dominates* B′, and we write B dom B′, if every path from the initial node to B′ passes through B. Note: every node dominates itself and the initial node dominates every node.

In the above example, B1 dominates B1, B2 and B3, while B2 and B3 dominate only themselves.

9.2.1 Natural loops

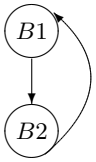
A *natural loop* in a flow graph is a subgraph which:

1. Contains a *header* – a node that dominates every other node in the subgraph.

2. Contains a *tail* – a node that is connected to the header and which has the property that all the other nodes in the subgraph are connected to it via a path that does not include the header.

Generally, to identify natural loops look for possible headers and matching tails.

For Example 1, B1,B2 is the only possible candidate pair, because there must be an edge from a node to a node that dominates it. Then include all other nodes connected to the tail via a path not including the head (in this case there aren't any).



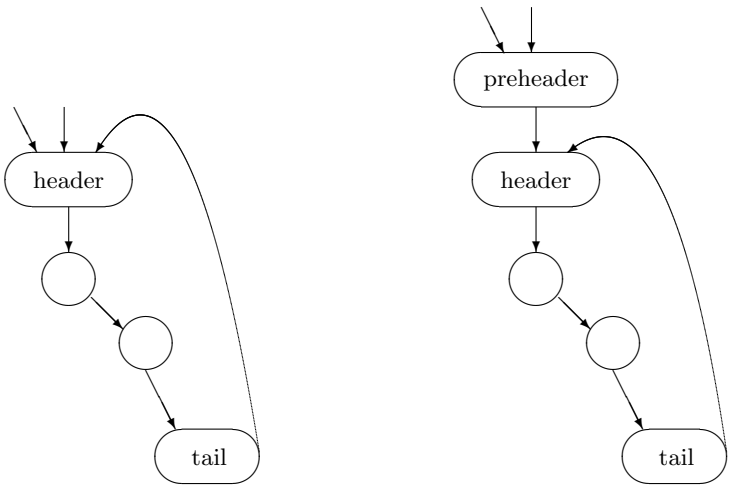
A natural loop is *strongly connected* – there is a path from any node to any other node in the loop. So, a natural loop *is* a loop in the sense that it is possible to go round and round forever, and there is a unique entry point (the header) to the natural loop from the rest of the flow graph.

A natural loop is an *inner loop* if no subgraph of it is a natural loop. So it contains no loops itself. When improving code, the best efficiency savings are likely to come from improving inner loops.

9.2.2 Code motion

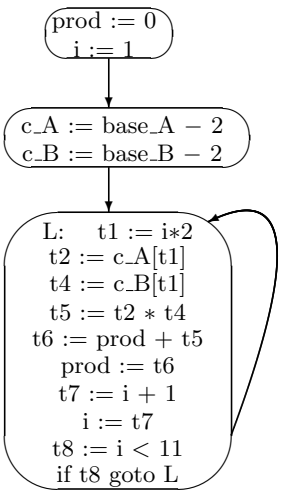
Code motion is a transformation that takes an expression whose value remains unaltered throughout the execution of a loop and places it outside the loop. The idea is that originally the expression is reevaluated each time the loop is traversed, but by taking it outside it is only evaluated when the loop is entered. We put such expressions in a preheader.

Given a natural loop, its *preheader* is a new block which is added to the flow graph immediately above the header of the loop. There is one new directed edge from the preheader to the header, and all the edges which used to enter the header, except the one from the tail, are now made to enter the preheader.



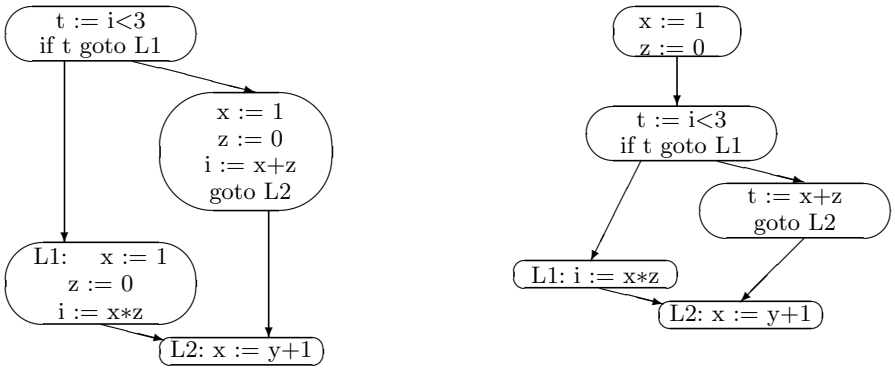
The code improver then places into the preheader any *loop invariants* - expressions from the loop whose value is constant within the loop.

In the partially optimized version of Example 2 the values of c_A and c_B are loop invariants. Notice however, that $t2$ may not be constant. It is always $c_A[t1]$ but the value of $t1$ may change during repeated executions of the loop, changing the value of $t2$. So it cannot be put into the preheader.



9.2.3 Code hoisting

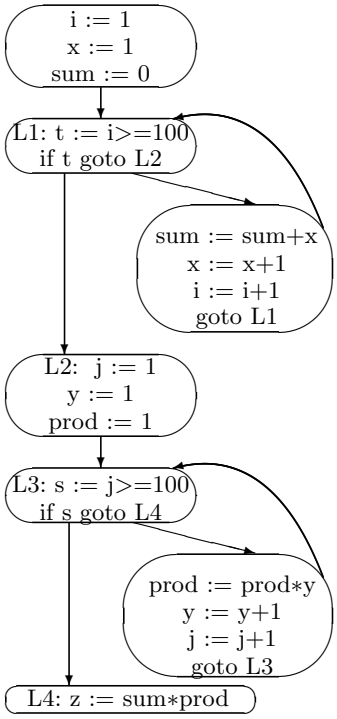
Expressions that appear in more than one basic block may be moved to a common ancestor block. If several lines of code occur in each block then separating these out can save memory space.



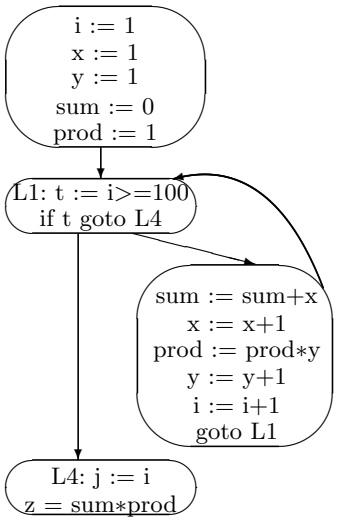
9.2.4 Loop fusion

Loops often have computational overheads which are nothing to do with the particular function of the loop, but are just 'book-keeping'. For example, a loop which is performing an iteration has to test and then increment the value of the iteration counter each time the loop is executed. If the code contains two loops it may be possible to merge them into one, so that there is only one set of such overheads instead of two.

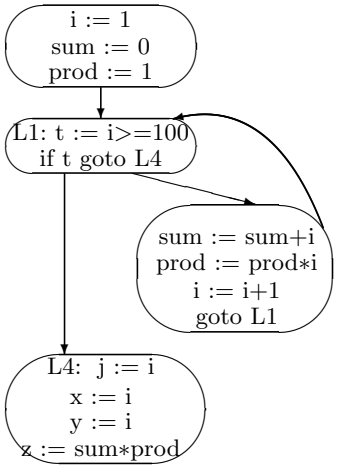
The following code contains two loops, the first calculating the sum of the integers between 1 and 100, and the second calculating their product.



We can merge the two loops, performing both calculations at the same time.

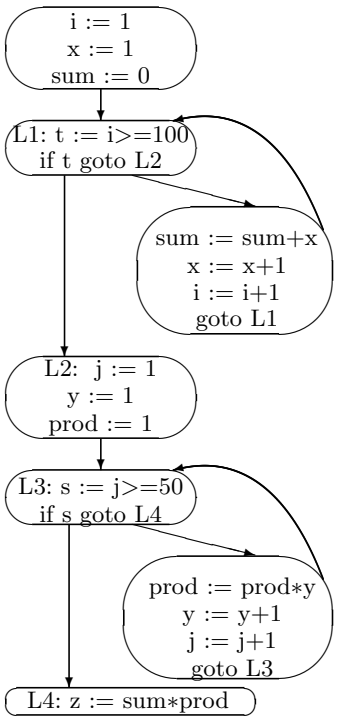


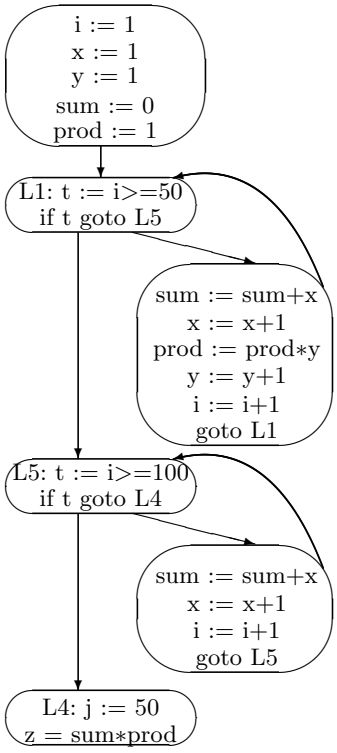
Of course, this code is still not optimal. We could do away with both x and y and use the counter i for the actual work of the loop as well.



The point in this section was simply to illustrate the efficiency gain by merging the actual loop overheads.

Even if the loops are iterating over different size sets we can still use loop fusion to gain some efficiency.





9.3 Directed Acyclic Graphs (DAGS)

In this section we shall look at how DAGs are used to help make structure preserving transformations of the types described in above. In particular, we shall use them to identify common subexpressions that can be eliminated.

A DAG is like a parse tree except that children can have more than one parent. The interior nodes of a DAG are labelled with operators, and the leaves are labelled with particular names from the three address code being optimized.

We construct a DAG for each basic block of the code, and associate an interior node with each statement in the block. (The same node may be associated with more than one statement.) There is a leaf of the DAG for a name if it is used in the block before a statement that redefines it is executed.

9.3.1 Constructing a DAG from code

We begin with the empty graph, and construct the graph from the leaves upwards, beginning with the first statement in the block and working down each in turn. We explain how to extend the graph for each possible type of statement.

1. Suppose that the next statement to be considered is of the form `x := y op z`. First we look to see if there are nodes that have y or z attached. If not we add new leaves labelled with y and/or z, and attach y and z, respectively, to these nodes. If there is already a node labelled *op* with children y, z we attach x to this node, otherwise we create a new node labelled *op*, attach x to it, and draw edges from the new node to

the two nodes y and z. Finally we look to see if there is any other node which has x attached. If there is we remove this x.

2. If the next statement to be considered is of the form `x := op z` we follow exactly the same procedure as (1) but ignoring the references to y.

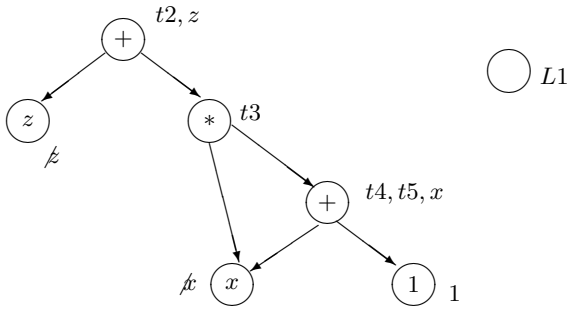
3. Suppose that the next statement to be considered is of the form `x := y`, we check to see if there is a node which has y attached. If there is not we add a new leaf labelled y and attach y to that. Then we just attach x to the node to which y is attached. Again we then look to see if there is any other node which has x attached, and if there is then remove it.

4. If the next statement is a relational operator such as `if z goto L`, we treat it as a statement of type (2) in which the LHS, x, is undefined. So we attach the label L to the node to which z is currently attached. (No problems arise here because each basic block can contain at most one goto statement, and this will be the last statement in the block.)

Example Consider the following block from Example 1 above:

```
t4 := x + 1
t3 := x * t4
t2 := z + t3
 z := t2
t5 := x + 1
 x := t5
goto L1
```

The associated DAG is:



9.3.2 Code improvements from DAGS

Constructing the DAG for a basic block automatically detects the common subexpressions. By looking at the DAG we can tell which names from outside the block are genuinely used within it, those being the names for which a leaf is created. If a leaf is not created the value of that name on entering the block is irrelevant. We can also tell by looking at the DAG which statements compute a value which still exists when the block is exited, being those statements whose associated node still has a name attached. These things help in the recognition of dead code.

In the above example, the values of $t2$ and $t5$ are not used outside the block so when we use substitution to replace them they become dead and hence can be eliminated.

9.3.3 Reconstructing code for DAGs

In order to be able to use DAGs we need to know how to reconstruct code from them. We do this by constructing three address statements at the nodes of the DAG. This is referred to as *evaluating* the node. When we have evaluated a node we assign the value to one of the names attached to the node.

We explain how to evaluate a node labelled x that has two children. Nodes with one child are evaluated in a similar way. By assumption the children will have already been evaluated, and if these nodes have more than one attached name one will have been singled out to carry the value of the node. If there is no attached name and the node is a leaf then the label of the leaf is taken to be its attached name. If the node is interior then a new temporary name is created and attached to the node. Thus we shall suppose that y, z are the chosen names attached to the children of the node. The node is now evaluated as $y\ op\ z$. We turn this into a three address statement by choosing a name which is attached to the node. This is then the name singled out to carry the value of this node. If the name chosen is x then the statement `x := y op z` is added to the list of code being constructed. We cannot single out x to be the chosen name if x also labels a child of another node that has not yet been evaluated. If this means that there is no legal choice of name then a new temporary name is created. If there is more than one legal choice of names then preference is given to those names that are still live (are used later) outside the block.

It is safest, although not entirely necessary, to re-evaluate the nodes in the order in which they were created. When code has been constructed for each node, except for the goto node if there is one, we add assignment statements for each name not singled out to carry the value of a node, but is still live at the end of the block, by adding statements `u := x` for each name u attached to the node whose value is carried by x. However, if u is dead at this point then we don't add `u := x` because it is dead code. Finally we add the code from the goto node, if there is one, which completes the reconstruction.

Example We reconstruct the code from the above DAG

```
t4 := x + 1     (can't choose x)
t3 := x * t4
 z := z + t3    (we can choose z since it is not needed
 x := t4        to evaluate another yet to be evaluated node)
goto L1
```

What would have happened if we had chosen x to carry the value of its node?

```
 x := x + 1
t3 := x * x   (if x is initially a, t3 has the value (a + 1)(a + 1)
              rather than a(a + 1) as in the original code.)
```

What if we had written the assignments first?

```
t4 := x + 1
 x := t4
t3 := x * t4   (same problem)
```

Notice The order of the children matters, y *op* z may not be the same as z *op* y.

Exercise Construct the DAG for the larger of the two basic blocks of the code given in Example 2 of Section 8. Use the DAG to reconstruct a more efficient equivalent block of code.

10 Error detection, reporting and recovery

The basic function of a parser is to check whether an input string is in the language defined by a particular grammar. In an ideal world, every string presented to a parser would be a valid string in that language, but as we all know, humans often present invalid strings to compilers. The usability of a real compiler is highly dependent on the quality of the feedback provided to the user in the case of an erroneous string being input. So far, our parsers effectively simply output YES or NO when given a string to check. A compiler that just stopped on an erroneous string and output NO would be almost unusable, because the users of the compiler would have to examine the string to find the source of the error: in effect they would have to simulate the behaviour of the parser themselves!

This section considers ways to augment parsers so that they provide useful feedback when errors are detected. We would like to do the following.

1. Report the *position* of an error accurately.
2. Issue a useful error message that explains the nature of the error in terms of the language being parsed, as opposed to the internal operations of the parser: few compiler users will be interested in the current state number the parser happens to be in, for instance.
3. Continue parsing the remainder of the input after the error has been detected so that further potential errors may be collected.
4. Not slow down the parsing process for strings which are in the language.

It turns out that there is a sort of inverse relationship between the power of a parsing technique and its ability to meet these criteria. LL(1) parsers expect to see a particular token at every stage of the parse, and so when an invalid token appears the parser immediately knows that an error has occurred, and it knows which tokens would have been valid at the error point. Thus criterion 1 is satisfied and it is relatively easy to fulfill criterion 2 because the parser knows what should have come next. A top-down, backtracking parser has much greater parsing strength but can only detect an error when all possible backtracking possibilities have been tested and failed. At this point the parser cannot know which particular partially consumed input string is the one that defines the error.

Criterion 3 is much more of a problem: our ability to continue cleanly after an error is as much a function of the grammar as the parsing technique. Modern block structured languages cause particular problems compared to earlier line oriented languages such as FORTRAN and BASIC. The reason for this is that in line oriented languages errors are rather localised: we know that a statement is terminated by a line end, so a sensible recovery technique is to simply throw away the rest of a line in which an error is detected.

10.1 Classes of error

Errors can be detected during various phases compilation.

- *Lexical errors* arise from strings of characters that can not even be built up into a sequence of tokens. An obvious example is the presence of a character that is not valid anywhere in the language: for instance the back quote character `` ` `` is illegal in any ANSI C program except when it occurs within a string literal, character literal or a comment. At a more complicated level, only a few sequences of punctuation symbols are local in ANSI C: the string `> >?` for instance is invalid.

- *Syntax errors* are detected by the parser and are caused by strings of well formed tokens that are not in the language. In this section we are mainly concerned with syntax errors and what to do after one is detected.

- *Semantic errors* arise from strings which are in the language but 'meaningless'. Standard grammars for ANSI C rely rather heavily on semantic checking: for instance the program fragment `1[3]` is syntactically permitted but meaningless because an integer constant cannot be indexed.

  A more frequently occurring kind of semantic error is the violation of a type rule, such as calling a function with too many parameters or attempting to assign a character literal to a string variable. Type checking is a basic feature of most modern programming languages, and all type checking falls into the class of semantic checking because as we have seen we cannot express type compatibility rules for an infinite language using a finite context free grammar.

  Reporting semantic errors can be difficult because in some compilers semantic errors are only detected after the initial parse has been completed and so the original source text may not be immediately available.

- *Logical errors* arise from a misunderstanding on the compiler user's part: either of the algorithm they are attempting to implement or of the semantics of the programming language. In either case the program passes through the compiler correctly but when executed displays behaviour other than that intended by the programmer. The compiler cannot help here: a better programmer must be obtained.

Sometimes a logical error can be caused by what is effectively a syntax error which converts the program into another valid program: consider this C fragment:

```
z=300;
if (a>b)
  while (z!=0);
  {
    y=b+z;
    z-=3;
  }
```

This innocuous looking while loop will in fact go into an infinite loop because of the misplaced semicolon after the while: the 'body' of the while loop looks like

a free standing code block. In cases like this a better programming language must be obtained: a language such as Modula-2 which explicitly delimits the end of each compound statement would be able to catch this error syntactically.

10.2 Error messages

Early compilers (and some badly written modern ones) issue error messages that convey little information to the user. This may be because they convey little information at all, or because the message is couched in terms that are meaningful to the compiler writer but not to the user. As an example of the latter: the `rdp` parser generator can be set up so that whenever a syntax error is detected the message tells the user which grammar production was being parsed at the time the error was detected. This is very useful to the compiler writer when debugging a grammar, but can only be confusing to the eventual compiler user who knows nothing of the detailed structure of the compiler's grammar, and most likely does not even know what a grammar is.

A common fault in compilers for small systems is to simply issue an error number rather than a full diagnostic. When memory was in very short supply, it was often difficult to fit the compiler and the compiled intermediate code into memory, and one of the first things to be jettisoned was the set of strings used to display the actual error messages. Instead, the user had to keep a sheet of error messages to hand whilst working and match up the displayed error numbers to the printed messages. This strategy is unnecessary on modern machines, but lazy implementors sometimes use very terse messages.

10.3 Error recovery

A compiler that simply stops when it first encounters an error is unacceptable in some environments, even if it issues an excellent error message. Ideally, we would like the compiler to carry on after the point at which the error was detected and look to see if there are any more errors in the source file.

Some errors are much easier to recover from than others. Semantic errors, in particular, tend not to disrupt the parse. This is not surprising: after all a semantic error is by definition embedded within a syntactically correct substring so all that needs to happen is for the error (say, a type violation) to be reported and for parsing to continue.

Some types of syntax error are easy to recover from too. Consider, for instance, a parameter list for a function that should be comma delimited, but which has had one of its delimiters mistyped as a semicolon. Once the parser has started to check a parameter list it should not expect to see any semicolons before the closing parenthesis, so it could, for instance, simply treat *any* punctuation symbol as a delimiter if it is not a close parenthesis (although, of course, an error message would be issued for non-commas).

Some types of syntax error are difficult to recover from. In particular errors relating to bracket nesting are particularly troublesome, be they block begin-end delimiters (such as `{ }` in C), arithmetic brackets or parameter list brackets.

The characteristic of these errors is that omission of a closing bracket changes the context of the remaining code. Consider the following fragments.

```
void b(void) {
  while (a>2) {
  if (a>b) {
    a = 3;
    if (a == 6) {
      z=5;
      y=4;
  } }
}
void a(int b) {}
```

The error here is in fact a missing brace `}` at the end of the second `if` statement, but the compiler will not notice the error until it gets to the start of the next function. In fact, depending on the error recovery strategy the definition of `void a()` may be deleted, and that will then cause later errors at points in the program where `a()` is called.

In free format block structured languages, bracketing errors can cause a cascade of error messages. A good recovery strategy ought to suppress these spurious errors, but in general it is extremely hard (if not impossible) to engineer such a strategy.

Error recovery from lexical errors is usually trivial: the lexer simply discards the characters that did not form a recognisable token, effectively converting them to white space. However, it may be that some valid tokens get discarded this way: should the string `&&>` be completely thrown away or converted to `&&`. So even lexical error recovery may cause knock on effects at the syntactic level.

10.4 Error correction

There have been attempts to engineer compilers that have error correcting properties. Sometimes these translators are called *do what I mean* systems. Imagine, for instance, a command line interface to an operating system that can do spelling correction on command names. The command `coppy` could easily be mapped to `copy`. At a more sophisticated level, humans often omit punctuation symbols such as ; and , from their programs, and in many cases simply inserting punctuation symbols can correct the programmer's error.

Quite sophisticated error correcting compilers were in vogue in the 1970's for use by students, on the basis that compiling was an expensive operation on large time sharing computers and anything that could be done to minimise the number of bad runs would be useful.

In practice error correcting compilers are rather dangerous: there is no guarantee that the 'corrected' program is the one that the programmer actually intended, that is simply converting a syntactically incorrect program into a syntactically correct one might still leave (or even create) semantic errors. In addition, real error correcting compilers do not actually manage to correct any but the most trivial errors.

10.5 Stop on first error

Given the enormous increase in parse speeds in recent years (as a result of technology improvements) it may not be necessary to bother with error recovery at all. A good example of the alternative approach is Turbo Pascal, a fast Pascal compiler for Intel architectures that combines an editor, debugger and compiler. The compiler source file is held in memory in the buffer, and code is written directly into memory. The compiler uses a single pass recursive descent parser, and can compile many thousands of lines per minute even on very low powered computers. As a result, the compiler can find the first error in a piece of code within a fraction of a second. Turbo Pascal simply stops at the first error with the editor showing the position of the error. The user then corrects this, and recompiles rather than getting a complete list of errors.

10.6 Panic mode error recovery

There is are large number of error recovery strategies in the literature, but no generally accepted best practice. The simplest and most easily implemented strategy is panic mode error recovery.

Panic mode recovery involves simply discarding input tokens until a token is seen that is in some supplied synchronisation set. Often synchronisation sets are defined on a rule-by-rule basis, so that we can talk about the set SYNC(N) where N is some nonterminal, in much the same way as we talk about FIRST(N) and FOLLOW(N). In fact, a good starting point for constructing a synchronisation set is to assign

$$\text{SYNC}(N_i) = \text{FOLLOW}(N_i) \cup \$$$

for all the N_i in the grammar.

Panic mode correction is simple, but will often cause large parts of the input to be skipped. Since these parts may include definitions and structure required in the postfix string (that part parsed after synchronisation is achieved) further errors may be triggered. In situations where multiple errors in statements are unusual it often works well and has the added advantage of being guaranteed not to go into an infinite loop.

Top down parsers lend themselves to panic mode error recovery because they match a complete grammar rule before returning. In effect, top down parsers guarantee to match a complete grammar rule at each invocation. A good error recovery strategy, therefore, is to leave the parser looking at a valid input token before returning from an invocation. In practice, this means that if we detect an error whilst parsing the grammar rule for nonterminal N we should at the very least ensure that the lexical analyser is supplying a token in FOLLOW(N) before returning.

11 Target specific code generation

In the previous section we have seen how code may be generated using attributes and semantic actions; and how standard optimisations may be applied to convert inefficient sequences of code into semantically equivalent pieces of code that either require less memory space or execute more rapidly. All of these techniques may be applied without reference to the actual machine on which the code may run (although in detail, some optimisations might only be applied on certain classes of machine).

After code has been improved in this general way we are left with the problem of emitting specific machine instructions that match our target architecture. There are three parts to this problem:

code selection The selection of code templates that implement program fragments on the target machine, for instance a `+` operator would normally map to an `ADD` instruction on the target hardware, but if the relevant expression were of the form

```
temp = temp + 1
```

there might well be a special `INC` (increment) instruction that could be used instead of the standard ADD instruction.

register allocation The selection of variables that should reside in machine registers. Register based variables are essentially available at zero time cost, as opposed to main memory variables which require a main memory cycle to be used during instruction execution. On most modern RISC architectures variables *must* be register resident to be used in an arithmetic or logic instruction, so register allocation is even more critical than on traditional memory-register architectures.

scheduling On processors with multiple functional units, or programmer-visible pipelines, operations must be carefully allocated to functional units and their ordering arranged so as to minimise structural, data and control flow hazards.

Code selection and register allocation are relatively well studied and relatively well understood problems. Code scheduling is a more recently studied area, mainly because in the 1970's some theoretical studies showed that the level of potential *instruction level parallelism* (ILP) within basic blocks was low – of the order of only two-three. Interest was renewed in the early 1980's when techniques developed for microcode compaction were applied to RISC like architectures with multiple functional units. Much of the current research in computer architecture and compiler design focuses on the interaction between the compiler and the architecture, and ways in which ILP may be exposed to the compiler and used to schedule code efficiently.

11.1 Code selection

In this course we have avoided dealing with specific architectures. This is because most traditional machines such as the VAX offer a wealth of special features which might be exploited by a compiler, but a study of such features does not generalise across different architectures. In an introductory course such as this we concentrate mainly on principles rather than the details of particular systems so a VAX specific code selector would be of little interest here.

With RISC architectures the situation is easier since the main commercial RISC architectures are much more similar to each other than any set of traditional architectures. For illustrative purposes, we shall construct an instruction set that reflects the main features of a real instruction set.

Instructions come in four main classes: arithmetic and logic operations such as `ADD` and `DIV`; copy instructions which load and store results to main memory or replicate data between registers; flow of control instructions including branch, jump, procedure call and procedure return; and system instructions such as those used to change the processors mode or manage interrupts. The system instructions are unlikely to be used by a compiler: they are more the province of the operating system than user programs.

The following is a simple three address instruction set for a processor which supports the four basic arithmetic operations along with branch-if-true, branch-if-false and branch-always instructions:

`ADD` *dst, src1, src2*	$dst \leftarrow src1 + src2$
`SUB` *dst, src1, src2*	$dst \leftarrow src1 - src2$
`MUL` *dst, src1, src2*	$dst \leftarrow src1 \times src2$
`DIV` *dst, src1, src2*	$dst \leftarrow src1 \div src2$
`EQ` *dst, src1, src2*	$dst \leftarrow src1 = src2$
`NE` *dst, src1, src2*	$dst \leftarrow src1 \neq src2$
`GT` *dst, src1, src2*	$dst \leftarrow src1 > src2$
`GE` *dst, src1, src2*	$dst \leftarrow src1 \geq src2$
`LT` *dst, src1, src2*	$dst \leftarrow src1 < src2$
`LE` *dst, src1, src2*	$dst \leftarrow src1 \leq src2$
`CPY` *dst, src*	$dst \leftarrow src$
`BEQ` *src, label*	if $src = 0$ then go to label
`BNE` *src, label*	if $src \neq 0$ then go to label
`BRA` label	go to label

In each case here, *dst, src1* and *src2* may be the name of a register or the address of a memory location. Note that this is in contrast to real RISC architectures in which only the `CPY` instruction has this generality with all other instructions taking only register names as operands.

Using this instruction set we can compile high level code fragments into assembly-like instructions. This high level program

```
a = b + c * d / e * f - ---g;
if a < 2 then begin z = 0; x = 3 end;
while x == 3 do x = x - 1;
```

generates this assembly level three address program:

```
 MUL  temp_1,c,d             ;temp_1 := c * d
 DIV  temp_2,temp_1,e        ;temp_2 := temp_1 / e
 MUL  temp_3,temp_2,f        ;temp_3 := temp_2 * f
 ADD  temp_0,b,temp_3        ;temp_0 := b + temp_3
 SUB  temp_5,0,g             ;temp_5 := 0 - g
 SUB  temp_4,temp_0,temp_5   ;temp_4 := temp_0 - temp_5
 CPY  a,temp_4               ;a := temp_4
IF_0:
 LT   temp_6,a,2             ;temp_6 := a < 2
 BEQ  temp_6,ELSE_0          ;ifn temp_6 go to ELSE_0
 CPY  z,0                    ;z := 0
 CPY  x,3                    ;x := 3
 BRA  FI_0                   ;go to FI_0
ELSE_0:
FI_0:
DO_1:
 EQ   temp_7,x,3             ;temp_7 := x == 3
 BEQ  temp_7,OD_1            ;ifn temp_7 go to OD_1
 SUB  temp_8,x,1             ;temp_8 := x - 1
 CPY  x,temp_8               ;x := temp_8
 BRA  DO_1                   ;go to DO_1
OD_1:
```

Note that many temporary variables are used which have the property that they are written to and read from exactly once. No variables from the high level program are written to again unless explicitly required by the high level program. This is to ensure that if a variable is still live (that is if it is used again further down) its value will still be available. In detail, many variables may be kept for longer than is absolutely required: no temporaries are reused here for instance. Data flow analysis techniques may be used to discover which variables have overlapping live ranges and a technique called *graph colouring* may be used to force variables with non-overlapping ranges to share storage.

11.2 Register allocation

Registers provide the processor with a small amount of memory that is accessible very quickly. Allocation of registers to the most frequently used variables can greatly speed up execution by reducing the number of main memory accesses. We can distinguish a family of register based machines.

- ◇ Accumulator based, with only one data register, eg the PDP-8 and very early microprocessors.
- ◇ Specialised registers for addressing (index registers) and data, eg the Motorola 6809 microprocessor with two data registers and two data registers.
- ◇ General purpose register machines such as the VAX-11, the MIPS, SPARC and Alpha RISC processors.

11.3 The register allocation problem

Register allocation breaks down into several phases:

1. Register counting wherein the number of registers used by an expression is calculated.

2. Register *allocation* proper in which we decide which variables will be put in registers.

3. Register *assignment* in which actual register numbers are associated with each registered variable.

11.4 Allocation schemes

Register allocation schemes vary considerably in complexity:

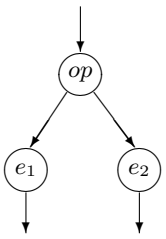
1. Registers are split into two groups, one for holding temporary results the other for storing variables. Allocation of registers for variables in inner loops is done first as this is where the speedup is greatest.

2. Usage counts allocate registers to the most frequently used variables.

3. Graph colouring uses the life time of variables to determine which variables and expressions can share registers. A node in a graph is created for each live value with edges between nodes representing overlapping live time intervals.

11.5 Register counting

Consider the following portion of an abstract parse tree:



The child nodes e_1 and e_2 each correspond to sub-expressions requiring n_1 and n_2 registers each to evaluate. If the left hand expression is evaluated first, then its result will have to be held in a register whilst the right hand side is evaluated, hence the total number of registers required is $\max(n_1,n_2+1)$.

If $n_1 < n_2 - 1$ then it makes sense to evaluate the right hand side first. Some compilers run through the tree swapping children for commutative operators (not subtract or division!) so as to minimise the 'natural' register count.

11.6 Programmer driven register allocation

In some languages (such as C or Bliss, a systems programming language used by DEC) register allocation can be done by the programmer by adding a `register` designator to declarations. Only do this after running a profile on your program!

There was a real compiler (for a language called PPL) which reordered expressions and then allocated registers in a first-come, first-served way. Only five registers were available, so heavily nested expressions caused the compiler to stop with a message asking the user to simplify the expression. This was not very satisfactory.

The process of selecting variables to be ejected from registers is called register *spilling* to main memory.

11.7 Usage counts

An easy way to decide which registers should be spilled is to count the number of times that a variable is referenced within basic block. If a variable is read again within a block after it has been written, then keeping that variable in a register will save one main memory cycle *per* read. If the block is a loop, then we also save a write at the end of the block. On the debit side, the register must be loaded at the start of the block (which costs two cycles), but if the block is loop then this negligible. For a block B loop that is executed many times this formula describes the benefits of registering variable x

$$\text{use}(x, B) + 2 \times \text{live}(x, B)$$

where $\text{use}(x, B)$ is the number of times x is read in B prior to being written and $\text{live}(x, B)$ is 1 if x is live on exit from B and 0 otherwise.

11.8 Register allocation by graph colouring

The purpose of a graph colouring phase is to detect variables with non-overlapping live ranges and force them to share storage.

A graph colouring register allocator uses two passes. In the first pass, the code generator assumes that there is an infinite number of registers available. In effect, the symbolic variable names in the intermediate code become register names. Stack pointers and other special variables are also assumed to have their own dedicated registers. Program temporaries, such as intermediate values required in address resolution for multi-dimensional arrays are also given dedicated registers.

The second pass attempts to map these symbolic registers onto physical registers in a way that minimises register spills (a spill being the unloading of a register into a main memory location so that the register may be freed for use by another variable).

The allocator constructs a register-interference graph in which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where the other is defined. A graph is said to be coloured if each node has been assigned a colour such that no two adjacent nodes have the same colour. An attempt is made to colour the graph using n colours where n is the number of registers available.

Graph colouring is NP-complete, but the following heuristic usually works well in practice: suppose a node m in a graph G has fewer than n neighbours.

Remove m and its edges from the graph to form a new graph G'. A colouring of G' can be extended to a colouring of G by assigning m a colour not assigned to any of its neighbours.

By repeatedly eliminating nodes having fewer than n edges from the register interference graph we either obtain the empty graph, in which case we can obtain a colouring of the graph by colouring the nodes in the reverse order in which they were removed, or we obtain a graph in which each node has n or more adjacent nodes in which case a colouring with n colours is impossible. At this point a node is spilled by introducing code to store and reload the register. A good rule is to avoid choosing spill nodes that are in inner loops.

After removing the spilled node from the graph, another attempt is made to perform a colouring: the whole process iterates until a successful colouring is obtained.

11.9 Spilling algorithms

Any register allocation scheme is critically dependent on the quality of the spilling decisions made. The Belladay register allocation algorithm is derived from work on page swapping algorithms

```
If all registers have not been allocated
     then allocate an unused one to X
else if any value in a register is no longer required
          then allocate this register to X
     else choose the register whose value is next used
            furthest away
          store the value of that variable if it has been
            changed since it was loaded
          allocate this register to X
```

This has the effect of spilling the register that will be unused for the longest period of time. There is an interesting analogy with paging schemes here. Recall that in a virtual memory environment page fault may trigger the ejection of a page from main memory onto disk, in other words a page may be spilled to the swap file. In operating systems a *least recently used* algorithm is often used so that the page that has remained untouched for the longest period up until now is ejected. Of course there is no way to be sure that a page that has recently been unused will not be needed again immediately but it is a useful heuristic.

Ideally, the operating system would like to know which page will remain unused for the longest time so as to minimise page thrashing, but since the operating system can not know what the future behaviour of processes will be it is forced to rely on history rather than prediction. A register allocator, on the other hand, has access to dataflow information which provides a reliable predictor of future behaviour. For straight line code (basic blocks) the allocator has exact information and can produce the optimal solution. For code containing loops the allocator usually assumes that the loop will be taken, since the probability of a loop block not being taken at all is rather low. For `if` statements

a conservative allocator will take the worst estimate provided by considering both branches equally likely.

We see here a spectrum from statically available information, such as the behaviour within a basic block, through fairly reliable estimation in loops and fairly unreliable estimation across `if` statements to the completely dynamic and unpredictable behaviour of processes within a system for which no source code is available for scrutiny.

11.10 Instruction scheduling and speculative execution

Just how rapidly could a program be completed? As a way of thinking about this problem, imagine that a program has executed once following a particular pattern of branches and loop counts and that we want to execute the same program again using exactly the same input data and that the previous execution path has been recorded in a *profile*. Assuming that the program is deterministic (and very few real computer systems are genuinely non-deterministic) then we would know in advance exactly which way each branch instruction would go. In effect, we could unroll loops and delete untaken branches so as to yield a single sequence of instructions – one very large basic block in fact. We could perform live range analysis and data dependency analysis to establish the degree of available instruction level parallelism in the block.

Now imagine that we have at our disposal an infinitely large computer. Here size is measured not just in terms of memory and disk capacity but (more importantly for our purposes) an infinite number of functional units each of which are capable of evaluating a result using three memory operands in a single cycle. We could *schedule* our monolithic program using as many units as are indicated by the level of ILP present at each point in the program. In effect, the maximum overlap would be provided and so the number of cycles required to complete the program would be minimised.

The real world is less convenient than this. To begin with, computers are finite. Even machines with multiple functional units may not be able to serve all levels of memory in a single cycle as we have seen. Most likely, a multi-functional unit will have a small number of registers that it can use efficiently and so we are left with a processor-memory bandwidth restriction just as in the traditional von Neumann bottleneck. Perhaps the critical restriction is that in reality we do not usually know the profile of a program in advance, so we have to make assumptions about the most likely path or *trace* which will be executed. We can schedule this path *speculatively*, that is assume that it will be taken but add enough house keeping code so that if another trace is taken then the work done speculatively can be undone. This is the basic idea behind trace scheduling, the first of a family of techniques that may come to dominate code generation.

The current frontier in computer architecture and compiler research is the blurring the line between dynamic and static analysis of code, either by adding hardware to support speculative execution or to use more sophisticated prediction algorithms to drive code schedulers. New architectures with multiple functional units are expected to be announced and these are likely to be al-

most impossible for humans to program directly. The wheel has turned full circle: early 1950's machines were hard to program because of the programmer visible timing constraints and now modern architectures present an array of capabilities that is too great to be kept track of without machine assistance.