CS24210 Syntax Analysis and Topics in Programming Languages

Edel Sherratt

About this Module

Your Aim: to become familiar with the concepts required for specifying and implementing programming lanuguages

Why?

Learning Outcomes – for the whole course

On successful completion of this course, you should be able to

- make effective use of compilers and other language processing software;
- apply the techniques and algorithms used in compilation to other areas of software engineering;
- understand the need for implementation independent semantics of programming languages;

How the course will work

- lectures, will require input from you
- scheduled practicals – absolutely essential
- reading – web based materials and library
- own practical work

About the Course

- Syllabus – www.aber.ac.uk/modules/current/CS24210.html
- The order will change
- The content may vary from year to year
- The examination will be based on the course *as taught*
- For the highest marks, evidence of additional reading, and more specially, *coherent thought* about the taught material will be demanded
- This means that your work should be paced
- Please point out any problems with the course content or presentation quickly – during the lecture, if necessary

Useful sources of information

- www.aber.ac.uk/~ dcswww/Dept/Teaching/Courses/CS24210
- It is not absolutely essential to purchase 'Bennett'; but you will need ready acces to some reasonably substantial book about compilation systems.
- Aho, Sethi and Ullman, *Compilers: Principles, Techniques and Tools,* ISBN 0-201-10194-7
- Fischer, C.N. and Leblanc, R.J. *Crafting a compiler with C,* ISBN 0-8053-2166-7 (or you may prefer the Java edition).
- Wirth, *Compiler Construction*, ISBN 0-201-40353-6

Language Processing Software

- Editors – e.g. nedit, gvim, vi, sed, ...
- Compilers and Interpreters – e.g. javac, gcc, perl, bash, csh ...
- Text formatters – e.g. LaTeX, nroff ...

...and the Languages they process

- gvim: ascii text, programming languages, markup languages
- javac: java programs
- gcc: ANSI C programs
- LaTeX: text marked up with latex formatting instructions
- html: text marked up with html formatting instructions
- perl: perl scripts
- bash: bash shell commands
- csh: C shell commands

How do we learn to use Language Processing Software Effectively?

- Learn all of these languages and software systems individually?
- In a 10-credit module – 80 hours of your time???!
- How do we cope with new languages and tools?
- How do we make sure our knowledge is good for life?

Lasting, transferable knowledge and skills in Language Processing

- We need some practice
- and also some abstract knowledge – transferable skills

Abstraction as a tool to support effective use of language processing software

- What kind of information do we need to keep in mind?
  depends on what we're thinking about
  - all languages: the fact that languages have structure;
  - Java: class structure;
  - perl: pattern structure, loop structure, ...
- What kind of information do we omit?
  - all languages: possible structures
  - Java: semicolons etc.
  - perl: * / < > . etc.
- Different abstractions for different purposes?
  - different levels of abstraction level (as above)
  - OR different perspectives (next slide)

Note - there are many other ways to use abstraction when thinking about formal language.

Three aspects of language

- syntax – form, shape, structure
- semantics – meaning
- pragmatics – use

Lexical errors in Java

What's wrong with

```
fruit = 7peach;
```

An identifier may not begin with a digit

Lexical errors in Java

How about

```
fred = _fred_value:
joan = _joan_value;
```

statement delimiter is ; not :

Lexical errors in Java

Or maybe

```
Abstract Class StackD {
  Abstract Object accept(StackVisitorI ask);
}
```

keywords 'abstract', 'class' and 'object' should contain only lowercase letters

Lexical errors in Java

Or even

```
my_counter = 7 * (loop_counter + 5o5);
```

the letter 'o' should not appear in a number

Lexical errors in Java

What elements of the Java programming language were affected by these lexical errors?

- identifier
- delimiter
- keyword
- number (unsigned integer)

What constitutes the lexis of a programming language?

identifiers, delimiters, keywords, numbers (of various kinds), comments, ...

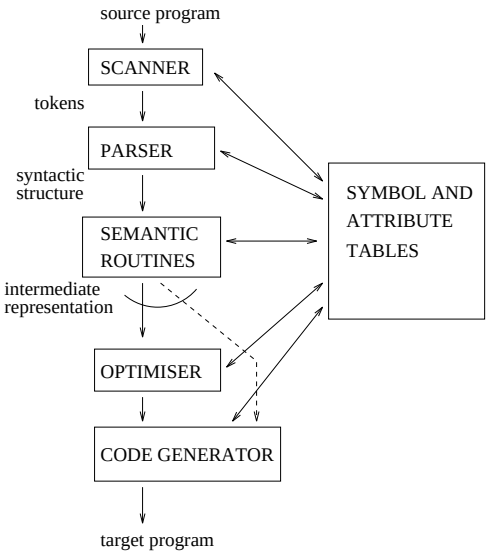
What part of the compiler catches lexical errors?

the scanner

How do we specify the lexis of a programming language?

We use regular expressions – we'll come back to this later.

Structure of a Compiler



Main Jobs of a Compiler

ANALYSIS of the Source Program

- lexical analysis
- syntactic analysis
- context checking

SYNTHESIS of the Target Program

- generation of intermediate representation
- optimisation
- code generation

The Scanner

The first job a compiler must do is to read the input character by character, and group the characters into useful bits called lexemes. This task is called lexical analysis.

Here are some lexemes:

␣for␣i␣in␣NumberofItems␣loop

␣ins␣:=␣42␣;␣

There are some nontrivial problems; illustrated by these FORTRAN lexemes:

␣2.E3␣

␣2␣.EQ.␣I␣

The Scanner

The lexemes are represented as tokens – often integers – to facilitate further processing by the compiler.

Tokens indicate the lexical class of the lexeme – identifier, reserved word, delimeter, ... &c. – and usually include a reference to the lexeme itself.

A typical encoding uses one or two digits to represent the lexical class of a token, with other digits acting as indices into tables where the lexemes themselves are stored.

For example,

ins := 42;

might be encoded as

0233 0076 0062 0011

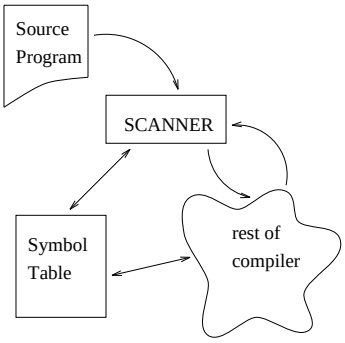
where the lowest order digit indicates the lexical class of the token, and the other digits index an entry for the token itself.

Other jobs done by the scanner

As well as doing its main job – recognising lexemes and encoding them as tokens – the scanner

- eliminates comments and "white space",
- formats and lists the source program, and
- processes compiler directives.

The Scanner in Context



Regular Expressions

Regular expressions provide a notation for describing the tokens used in modern programming languages.

A regular expression specifies a pattern. The set of character strings that match a regular expression is called a regular set.

For example,

$(underscore|letter)(underscore|letter|digit)*$

is a regular expression that specifies the set of Java identifiers – provided we have defined 'underscore', 'letter' and 'digit'.

Regular expressions and what they match

Conventional notation	Unix notation	matches
ϕ		no string
ϵ		the empty string
s	s	the string s
$A\|B$	$[AB]$	any string that matches A or B
AB	AB	any string that can be split into a part that matches A followed by a part that matches B
$A*$	$A*$	any string that consists of zero or more A's

s is any character string

A and B are regular expressions

Shorthands

- $A+$ one or more A's
  equivalent to $AA*$
- $A?$ zero or one A
  equivalent to $[\epsilon A]$
- $[0-9] = [0123456789]$

Constructing regular expressions

Write a regular expression that matches the Java keyword **class**

Constructing regular expressions

Using conventional notation, extend your regular expression so that it matches any of the Java keywords: **abstract class extends**

Write an equivalent regular expression using Unix notation.

Constructing regular expressions

Using conventional notation, write a regular expression that matches unsigned integers

Do the same, this time using Unix notation.

Constructing regular expressions

Extend each of your previous regular expressions so that they match signed integers.

More information

To find out all about regular expressions in Unix, see

```
man -s 5 regexp
```


Using Regular Expressions

- Unix applications
  - substitution using gvim
    $< range > s/ < regexp > / < substitution >$
  - searching files for patterns
    $grep \ < regexp > < path >$
- Specifying valid input

```
month   (JAN|FEB|MAR|APR|MAY|JUN|JUL|
            AUG|SEP|OCT|NOV|DEC)
day     [1-9]|([12][0-9])|(3[01])
year    ([19][0-9][0-9])|2000
```

- perl pattern specifications
  $t.e$ matches the or tee but not $tale$

Unix Pattern Matching

- There are inconsistencies between the ways different characters are used in different Unix applications and situations.
- In particular, characters are treated differently by the shell in filename expansion and by applications for pattern matching.
- For example, * means match 0 or more of the preceding expression in vi, gvim, sed, awk, grep and egrep, but * means match any string of characters in filename expansion

Unix Pattern Matching

- You can use single quotes (or, in some cases, double quotes) to bypass the shell and pass special characters to an application.
- Look up 'Unix in a Nutshell', section 6, Pattern Matching, for more detail
- Alternatively, look up the man pages for the applications you are using (man vi will tell you all you need for gvim).

Perl - Practical Extraction and Report Language

- scanning text files
- extracting information
- printing reports
- features of C, sed, awk and sh
- When sed, awk and shell scripts are not quite enough ...
- ...and C is a bit too much
- ...Perl is likely to be just right.

Perl example – derived from examples by Lynda Thomas

```
manuel% more read_a_file.pl
#!/usr/local/bin/perl

print "Filename? ";
$file = <STDIN>;        # read filename
open(FILE,$file);       # open the file
@file_content = <FILE>; # read into an array
close(FILE);
print @file_content;    # print the array
```


Perl example

```
manuel% cat names
lynda dave tom rory lynda rory lynda

manuel% read_a_file.pl
Filename? names
lynda dave tom rory lynda rory lynda

manuel% ls -l read_a_file.pl
-rwxr-----   1 eds      staff
    264 Oct 13 14:33 read_a_file.pl
```

Perl identifiers and types

```
manuel% cat trythis.pl
#!/usr/local/bin/perl

$scalar_string_id = "this is a string";
print $scalar_string_id . "\n";

$scalar_number_id = 7;
print "scalar_number_id = $scalar_number_id\n";

@array_id = ("peach","fig","apricot");
print @array_id[0].@array_id[2].@array_id[1]."\n\n";

%hash_id = ("peach", 2, "fig", 8, "apricot", 3);
print "We have ".@hash_id{"peach"}." peaches\n";
print "We have ".@hash_id{"fig"}." figs\n";
print "We have ".@hash_id{"apricot"}." apricots\n";
```


Perl identifiers and types

```
manuel% trythis.pl
this is a string
scalar_number_id = 7
peachapricotfig

We have 2 peaches
We have 8 figs
We have 3 apricots
```


Perl loops – from an example by Mike Slattery and Lynda Thomas

```
manuel% cat loopy.pl
#!/usr/local/bin/perl

while (<>) {
  @words = split;
  foreach $w (@words) {
    $count{$w}++;
  }
}

@sortedkeys = sort keys(%count);

foreach $w (@sortedkeys) {
  print "$w\t$count{$w}\n";
}
```


Perl loops

```
manuel% cat names
lynda dave tom rory lynda rory lynda

manuel% loopy.pl names
dave    1
lynda   3
rory    2
tom     1
```

Pattern matching in Perl

```
manuel% cat pattern.pl
#!/usr/local/bin/perl

while (<>) {
  @words = split;
  foreach $w (@words) {
    $count{$w}++;
  }
}

@sortedkeys = sort keys(%count);

foreach $w (@sortedkeys) {
  print "$w\t$count{$w}\n" if ($w =~ m/^l/);
}
```


Pattern matching in Perl

```
manuel% cat names
lynda dave tom rory lynda rory lynda marilyn

manuel% pattern.pl < names
lynda   3
```


A Perl subroutine that uses pattern matching

```
manuel% cat subroutine.pl
#!/usr/local/bin/perl

# from a program by Mike Slattery, June 1996

# swap takes a string, and replaces any words in %table
# by corresponding replacements

%table = ('i','you', 'you','i', 'my','your',
   'your','my');

sub swap {
  local ($in) = @_;
  local ($w, $head, $tail, $new);

  if ($in =~/[a-z]+/) {
    $w = $&;    # the match
    $head = $`; # before the match
    $tail = $'; # after the match

    # look up $new in %table; if not found, $new = $w
    $new = $table{$w} || $w;

    # put the sentence back together
    $head.$new.&swap($tail);
  }

  else { $in }
}
```


...and the main program

```
while ($input = <>) {
   chop $input;
   $input =~ tr/A-Z/a-z/;
   print &swap($input)."\n";
}
```

Running the program

```
manuel% subroutine.pl
i hate computers
you hate computers
i like running
you like running
you are daft
i are daft
i'm daft too
you'm daft too
i love computers really
you love computers really
^D
```


Perl modules

Perl modules encapsulate useful functions so that they can be made available to other Perl programs.

If you want to use the functions in a Perl module, you place a line calling the `use` function near the top of your program:

For example, provided the CGI module has been installed on your system,

```
use CGI;
```

will make the functions in the CGI module available to your program.

Example – Hello web-world

```
#!/usr/local/bin/perl

use CGI qw/:standard/;

print header;
print start_html, "Hello web-world!", end_html;
```


Example – Hello web-world!

If this program is in

```
/aber/eds/cgi-bin/hello_world.pl
```

then going to the url

```
http://users.aber.ac.uk/cgi-bin/user/eds/hello_world.pl
```

will cause it to print 'Hello web-world!' to the browser window.

A more interesting form – processForm

```
#!/usr/local/bin/perl

use CGI qw/:standard :html3/;

print header, title("form.pl");

sub processForm {
# this is done if the 'submit' button has been pressed

  print start_html,

    "You, ".param('name')."., whose favorite colors are ";

    foreach (param('colour')) {print "$_, " }

    print " are on a quest which is '", param('quest'),
    "', and are looking for the weight of an ",
    param('swallow'),
    ".  And this is what you have to say for yourself:",
    p, param('text'),

    hr,
    "And here is a list of the parameters you entered...",
    dump();

  end_html;
}
```


A more interesting form – printForm

```
sub printForm {
# this is run if 'submit' has not been pressed

  local @colour = qw/chartreuse azure puce cornflower
    olive opal mustard/;

  local @swallow = ("African Swallow",
    "Continental Swallow");

  print start_html,
    h2 ("Pop Quiz"),
    start_form,
      "What is thy name:",
         textfield(-name=>'name',-size=>20),

      p, "What is thy quest:",
         textfield(-name=>'quest',-size=>20),
```



```
      p, "What is thy favorite colour:",
      checkbox_group (-name=>'colour',
        -values=>\@colour),

      p, "What is the weight of a swallow:",
      radio_group (-name=>"swallow",
        -values=>\@swallow),

      p, "What do you have to say for yourself",
      textarea(-name=>'text',-rows=>5,-cols=>60),

      submit(-name=>'submit',
        -value=>"Press here to submit your query."),
    end_form,
    hr,

    "CGI.pm version of Steven E. Brenner's combined form",
  end_html;
}
```

A more interesting form – main program

```
if (param('submit')) {
  processForm;
} else {
  printForm;
}
```

Perl - some url's

- http://language.perl.com/
- http://www.perl.com/pace/pub
- http://www.perl.org/

Finite State Automata

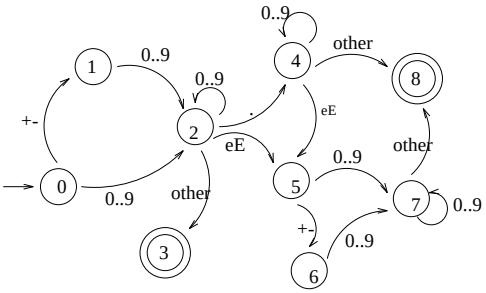
A finite state automaton (FSA) recognises strings that belong to a regular set.

It consists of

- a finite set of states,
- a set of transitions from state to state,
- a start state and
- a set of final states, called "accept states".

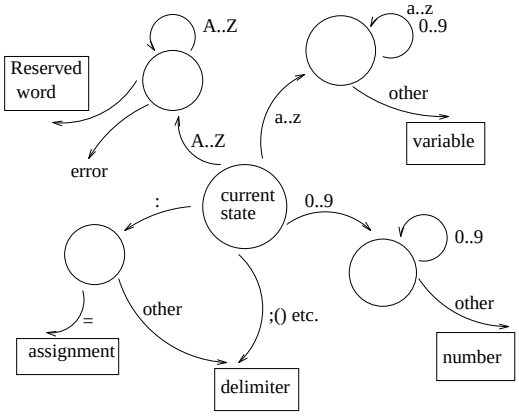
A FSA can be represented by a transition diagram – a directed graph whose nodes are labelled with state symbols, and whose edges are labelled with characters.

FSA that recognises numbers



This FSA is deterministic – for a given state and input character, the next state is uniquely determined.

Fragment FSA for a typical programming language



Pseudo code for a scanner

Assume one character, 'ch', has been read by 'getch', which reads a character, skips white space and produces a listing.

```
case ch of:
    ';' : token := semi
          getch
    ':' : getch; if ch = '='
                  then token := assign
                       getch
                  else token := colon
    '0' .. '9' : token := number
                 while ch in ['0' .. '9']
                       getch
                 endwhile
    ... &c.
```


Transition Tables

In general, a deterministic FSA can be programmed as a language independent driver that interprets a transition table. The transition table for the FSA that recognises numbers looks like this.

State	Character						
	+	-	.	e	E	0 ... 9	other
0	1	1				2	
1						2	
2			4	5	5	2	3
3							
4				5	5	4	8
5	6	6				7	
6						7	
7						7	8
8							

A scanner driver

```
State:= InitialState;
Read(CurrentChar);
loop
    NextState := TT(State,CurrentChar);
    exit when NextState = Error or
              CurrentChar = EOF;
    State := NextState;
    Read(CurrentChar);
end loop
if State in FinalStates
    then return valid token
    else lexical error
end if;
```


Using a Scanner Generator

There are algorithms for translating regular expressions to nondeterministic finite state automata (NFA), and for translating NFA to deterministic finite state automata (DFA).

These are used in scanner generators like lex, developed by M.E. Lesk and E. Schmidt of Bell labs.

Programming effort is limited to describing lexemes – an example of declarative programming.

Given input consisting of character class definitions, regular expressions with corresponding actions, and subroutines, lex generates a scanner called yylex.

Symbol Tables

The compiler uses symbol tables to collect and use information about names that appear in a source program.

Each time the scanner encounters a new name, it makes a symbol table entry for the name. Further information is added during syntactic analysis, and this information is used (and further extended) during semantic analysis and code generation.

Symbol Tables

The kind of information stored in a symbol table entry includes items like

- the string of characters denoting the name,
- the block or procedure in which the name occurs,
- attributes of the name (e.g. type and scope),
- the use to which the name is put (e.g. formal parameter, label ...)
- parameters, such as the number of dimensions in an array and their upper and lower limits, and
- the position in storage to be allocated to the name (perhaps).

The Symbol Table as an Abstract Data Type

Conceptually, a symbol table is a collection of pairs:

(name,information)

with operations to

- determine whether or not a name is in the table,
- add a new name,
- access the information associated with a given name,
- add new information for a given name, and
- delete a name, or a group of names.

Symbol Table Implementation

Symbol tables are typically implemented using

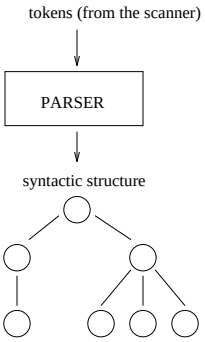
- linear lists,
- hash tables, or
- various tree structures.

In selecting an implementation, speed of access is traded against structural complexity.

The Scanner and the Parser

- What does the scanner (lexical analyser) do?
- What other kinds of analysis needs to be done?
- What parts of a compiler do these other kinds of analysis?

The Parser



Tasks Performed by the Parser

- Recognising syntactic structure.
- Verifying correct syntax.
- Handling errors.
- Building the syntax tree.

Defining the syntax of a programming language

The parser is driven by a formal description of the syntax of the language it parses.

Regular expressions are not sufficient to describe the structures found in programming languages.

For example, consider

```
for ... loop
    ...
    for ... loop
        for ... loop
            ...
        endloop
    endloop
    ...
endloop
```

Regular expressions cannot define nested constructs.

Context Free Grammars

Context free grammars provide a way of specifying the syntactic structure of modern programming languages.

A context free grammar contains rules.

```
<expr> ::= <expr> + <term> | <term>
<term> ::= <term> * <factor> | <factor>
<factor> ::= id | number | (<expr>)
```

This notation is called Backus-Naur Form (BNF)

Context Free Grammars

The grammar rules contain terminal and nonterminal symbols. The terminal symbols are the symbols of the language being parsed, while the nonterminal symbols are used in the grammar to represent whole phrases of the language.

Each rule has a left side consisting of a single nonterminal symbol. (This is what is meant by "context free").

Each rule has one or more alternative right sides; each right side consists of a string of terminal and nonterminal symbols.

One of the nonterminal symbols is defined to be the start symbol; in the example, the start symbol is $< expr >$.

Alternative forms of BNF

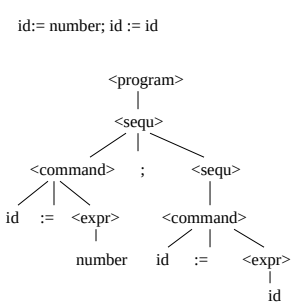
BNF can be simplified in various ways.

```
PROGRAM -> SEQU
SEQU    -> COMMAND | COMMAND ; SEQU
COMMAND -> id := EXPR |
   if EXPR then SEQU else SEQU endif |
   while EXPR do SEQU endwhile
EXPR    -> number | id
```

```
PROGRAM -> SEQU
SEQU    -> COMMAND [; COMMAND]*
COMMAND -> id := EXPR | ... etc.
```


Parse Trees

The derivation of a string from a grammar can be represented by a parse tree.



Approaches to parsing

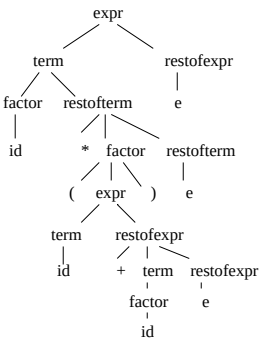
- Top down parsing – start with start symbol and build tree from the root down
- Bottom up parsing – start with input tokens and build tree from the leaves up

Parse Trees and Abstract Syntax Trees

Consider the grammar

```
expr ::= term restofexpr
restofexpr ::= + term restofexpr | e
term ::= factor restofterm
restofterm ::= * factor restofterm | e
factor ::= number | id | (expr)
```

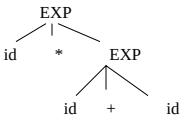

A Parse Tree for id * (id + id)



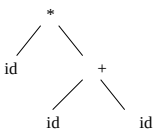
This parse tree is cumbersome, and not necessarily well suited to the later stages of compilation.

Abstract Syntax Trees for id * (id + id)

An abstract syntax tree for id * (id + id) could look like this



or like this



Abstract syntax trees are compact, and adapted to the needs of the later stages of compilation.

Different Parsing Methods

- top-down parsing with backtracking
- top-down deterministic parsing
  – e.g. LL(K), LL(1)
- bottom-up deterministic parsing
  – e.g. LR(K), LALR(K), LALR(1)
- operator precedence parsing,
  a bottom up parsing method that is especially useful for expressions like $a + b * c$

Usually, the grammar has to be restricted in some way to make each of these parsing methods work.

For example, top down parsing methods cannot deal with left-recursive grammars:

```
E ::= E + T | T
T ::= T * F | F
F ::= (E) | id
```

Instead, the grammar should look like this for top-down parsing:

```
E  ::= T E'          E' ::= + T E' | e
T  ::= F T'          T' ::= * F T' | e
F  ::= (E) | id
```

where e stands for the empty string.

Recursive Descent Parsing

Recursive descent parsing is very easy to implement.

For each nonterminal, a routine is written to parse all the strings that can be derived from the nonterminal. This means that the program corresponds closely to the grammar.

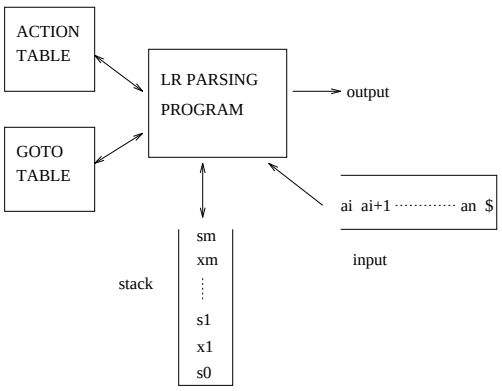
For example, given the rule

```
expr -> term restofexpr
```

our pseudocode might be

```
parse_expr = parse_term;
             parse_restofexpr;
             return tree
```


LR(1) Parsing



a1, ..., *an* are terminal symbols
$ is an end of file marker
x1, ..., *xm* are grammar symbols (terminals or nonterminals or both)
s0, ..., *sm* are state symbols

How the LR(1) Parser Works

1. It determines *sm*, the state symbol on top of the stack, and *ai*, the current input symbol.

2. It then consults the action table entry for *sm* and *ai*,

   action[sm,ai]

   which has one of the values

   shift s, reduce A ::= b, accept, or error.

3. Then it carries out the prescribed action, after which it will repeat the whole process, or terminate.

The LR Parsing Algorithm

```
a:= scan (input) /* first token */
REPEAT FOREVER
  s := state on top of stack
  IF action[s,a] = shift s'
     THEN push a onto the stack
          push s' onto the stack
          a := scan(input) /* next token */
  ELSEIF action[s,a] = reduce A ::= b
     THEN pop (2 * |b|) symbols off stack
          s' := new state on top of stack
          push A onto the stack
          push goto[s',A] onto the stack
          output rule A ::= b
  ELSEIF action[s,a] = accept
     THEN exit successfully
  ELSE /* action[s,a] = error */
     execute error routine
END REPEAT
```


Example

Consider the expression grammar

```
Rule 1:     E ::= E + T
Rule 2:     E ::= T
Rule 3:     T ::= T * F
Rule 4:     T ::= F
Rule 5:     F ::= (E)
Rule 6:     F ::= id
```

Suppose we want to parse the expression

```
id * id + id
```


Parsing Action Table

State	Input Token					
	id	+	*	(	)	$
0	s5			s4		
1		s6				accept
2		r2	s7		r2	r2
3		r4	r4		r4	r4
4	s5			s4		
5		r6	r6		r6	r6
6	s5			s4		
7	s5			s4		
8		s6			s11	
9		r1	s7		r1	r1
10		r3	r3		r3	r3
11		r5	r5		r5	r5

si – shift and stack state i
rn – reduce using rule number n
accept – accept
blank – error

Goto Table

State	Nonterminal E	T	F
0	1	2	3
1			
2			
3			
4	8	2	3
5			
6		9	3
7			10
8			
9			
10			
11			

Parsing $id * id + id$

Stack	Input	Action
0	id*id+id$	shift 5
0id5	*id+id$	reduce F ::= id
0F3	*id+id$	reduce T ::= F
0T2	*id+id$	shift 7
0T2*7	id+id$	shift 5
0T2*7id5	+id$	reduce F ::= id
0T2*7F10	+id$	reduce T ::= T*F
0T2	+id$	reduce E ::= T
0E1	+id$	shift 6
0E1+6	id$	shift 5
0E1+6id5	$	reduce F ::= id
0E1+6F3	$	reduce T ::= F
0E16+T9	$	reduce E ::= E+T
0E1	$	accept

Shift-Reduce and Reduce-Reduce Conflicts

Suppose a grammar included a rule like

```
T ::= X Y | X.
```

Having read an X, the parser could reduce it to a T, or could shift more symbols that might be a Y. This is called a *shift-reduce conflict*.

Another kind of conflict is know as a *reduce-reduce conflict*. This occurs when the parser cannot decide which of several possible productions to apply in reducing.

If it is impossible to generate LR(1) action and goto tables for a grammar, then the grammar is said to be not LR(1). Fortunately, most programming languages can be defined using LR(1) grammars.

LALR Parsing

A grammar for a typical programming language might have 50 to 100 terminals and about 100 productions (grammar rules). The LR(1) tables for such a language would have thousands of states!

In practice, a technique called LALR is used, giving rise to several hundred states for a language like Pascal.

LALR(1) is slightly more restrictive than LR(1), but experience has shown it to be adequate for the kind of languages we want to parse.

yacc is an LALR parser generator which was written by S.C. Johnson in the early 1970s.

Building the Syntax tree

The parser builds the abstract syntax tree by invoking a tree building routine each time a grammar rule is applied.

The tree building procedures are defined using an extended grammar, called an *attribute grammar*.

Building the Syntax tree

For example, if we were using an LR parser to parse expressions, we might use an attribute grammar like this to build the syntax tree:

```
E ::= E1 + T {E.treeptr :=
              mknode(E1.treeptr,'+',T.treeptr)}
E ::= T      {E.treeptr := T.treeptr}
T ::= T1 * F {T.treeptr :=
              mknode(T1.treeptr,'*',F.treeptr)}
T ::= F      {T.treeptr := F.treeptr}
F ::= (E)    {F.treeptr := E.treeprt}
F ::= id     {F.treeptr := mkleaf('id')}
```

where mknode returns an internal node of the syntax tree, and mkleaf returns a leaf node.

Building the Syntax tree

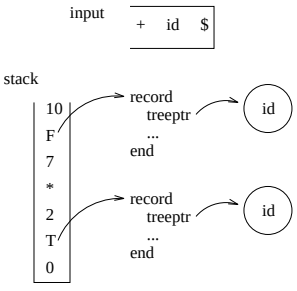
This attribute grammar deals with a single attribute, treeptr, a pointer to a fragment of the syntax tree.

treeptr is an attribute of E, T and F.

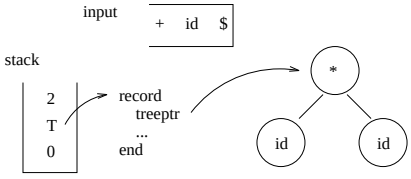
treeptr is called a *structural attribute*, because it contains information about the structure of the abstract syntax tree.

During a bottom up parse of $id * id + id$, using the parsing action and goto tables shown previously, the stack, input, and syntax tree would appear as illustrated.

Building the Syntax tree



ACTION: reduce T::= T1 * F {T.treeptr := mknode T1.treeptr, *, F.treeptr}
GOTO(0 , T) = 2



Recall the grammar

```
PROGRAM -> SEQU
SEQU    -> COMMAND | COMMAND ; SEQU
COMMAND -> id := EXPR |
   if EXPR then SEQU else SEQU endif |
   while EXPR do SEQU endwhile
EXPR    -> number | id
```

Give some examples of programs written in this language.

Draw parse trees for your programs.

Tree building routines

What kind of abstract syntax trees do we want?

What kind of tree building routines should be added to our grammar to build these?

```
PROGRAM -> SEQU

SEQU    -> COMMAND
    | COMMAND ; SEQU

COMMAND -> id := EXPR

    | if EXPR then SEQU else SEQU endif

    | while EXPR do SEQU endwhile

EXPR    -> number | id
```


Tree building routines

```
PROGRAM -> SEQU                        {
        PROGRAM.treeptr = SEQU.treeptr }

SEQU    -> COMMAND                     {
        SEQU.treeptr = COMMAND.treeptr }

  | COMMAND ; SEQU                     {
        SEQU.treeptr = makenode(sequ,
          COMMAND.treeptr, SEQU.treeptr); }

COMMAND -> id := EXPR {
        COMMAND.treeptr =
          makenode(':=',makeleaf(id), EXPR.treeptr); }

  | if EXPR then SEQU else SEQU endif   {
        COMMAND.treeptr = makenode(if,EXPR.treeptr,
          makenode(branch, SEQU1.treeptr,
            SEQU2.treeptr);}

  | while EXPR do SEQU endwhile         {
        COMMAND.treeptr =
           makenode(while,EXPR.treeptr,SEQU.treeptr); }

EXPR    -> number   {EXPR.treeptr = makeleaf(number)};

  | id              {EXPR.treeptr = makeleaf(id);}
```


Static Checking

(also known as "context checking" or "semantic analysis")

- type checking
- flow of control checks
- uniqueness checks
- name related checks

Static Checking by Attribute Evaluation

An attribute grammar describes static checks.

Attributes used in static checking are called *semantic attributes*.

The abstract syntax tree (AST) is traversed and decorated with attributes until it contains enough information so that static checks can be made.

Static Checking

Static checking can be performed after the AST has been constructed, or "on-the-fly", while it is being built.

In the first case, the AST is traversed, usually more than once, and attributes are evaluated each time a node is visited.

On-the-fly evaluation methods operate in conjunction with the parser (just like the tree building routines). They are used for single pass compilation.

The parsing method used limits the kind of attribute flow that can be accommodated when on-the-fly evaluation is carried out.

Intermediate Representations

The "front end" or analysis phases of a compiler produce intermediate code, which is processed by the code generator or "back end" of the compiler.

Three kinds of intermediate representation are

- abstract syntax trees,
- postfix notation and
- 3-address code.

3-Address Code

3-Address Code is like assembly language. It consists of a sequence of statements like

```
x := y op z
```

where x, y and z are names, constants, or temporary names generated by the compiler and op is an operator.

Statements can have labels, and there are statements that modify flow of control. For example:

```
label: x := y op z
goto label
if x relop y goto label
```

3-Address Code

Example: 3-address code corresponding to the syntax tree shown earlier.

```
t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a  := t5
```


Translation into 3-Address Code

Syntax directed translation of the source program into 3-address code can be defined using an attribute grammar.

Example: An attributed grammar rule for translating a 'while' loop to 3-address statements:

```
S ::= while E do S1
  {S.begin := newlabel;
   S.after := newlabel;
   S.code  :=
     S.begin <> ':' <> E.code <>
     'if' <> E.place <> '= 0' <>
     'goto' <> S.after <>
     S.code <>
     'goto' <> S.begin  <>
     S.after ':'
  }
```


3-address code for computing a scalar product – an example from Aho, Sethi and Ullman

Assume a and b are two vectors of length 20. The following 3 address statements compute the scalar product of a and b. (It assumes that the target machine has four byte words).

```
(1)  prod := 0
(2)  i := 0
(3)  t1 := 4 * i
(4)  t3 := a[t1]
(5)  t4 := b[t1]
(6)  t5 := t3 * t4
(7)  prod := prod + t5
(8)  i := i + 1
(9)  if i <= 20 goto (3)
```


Code Generation

Code generation is the final phase in compilation. From an intermediate represention of the source program, the code generator produces a target program.

The code generator's main tasks are

- memory management,
- register allocation and
- instruction selection.

Code Generation

The code that is generated should be

- compact,
- fast and
- effective in its use of machine resources.

Memory Management

The code generated by the code generator implements a memory management strategy. This means determing

- when variables are to be bound to storage, and
- where storage is to be allocated in memory.

Variables appear as names in 3-address code. Each name in a 3-address statement refers to a symbol table entry. The entry includes information about the type of the named data item. The code generator uses this type information to produce code that allocates a suitable amount of space for the data item.

Register Allocation

Register allocation entails

- selecting the variables that will reside in registers, and
- choosing the specific register in which each variable will reside.

Register allocation must be well done if the target code is to be compact and fast.

Instruction Selection

Instruction selection means choosing suitable target machine instructions to implement the intermediate program.

For example, suppose the target machine includes an instruction

```
INC
```

to add one to a register. Then

```
INC R0
```

is a better translation of a := a + 1 than

```
MOV a, R0
ADD #1, R0
MOV R0, a
```

Flow of Control Information

The code generator uses information about the flow of control of the intermediate program to produce efficient target code.

This flow of control information is represented as a flow graph.

Nodes in the flow graph represent *basic blocks* of instructions, in which control passes sequentially from instruction to instruction.

Edges in the flow graph represent transfers of control.

Basic Blocks

In a 3-address program, a basic block is a sequence of 3-address statements in which control begins at the first instruction, and passes through to the last instruction without halting or branching.

For example,

```
t1 := a * a
t2 := a * b
a  := t1 + t2
```

is a basic block.

A basic block computes a set of expressions. These expressions are the values of the names that are *live* on exit from the basic block.

Flow Graphs for 3-address Programs

A flow graph of a 3-address program has basic blocks of 3-address statements as its nodes.

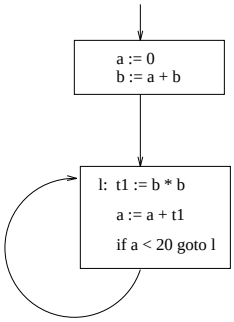
The node that contains the first statement of the program is called the initial node.

Suppose B1 and B2 are nodes. There is a directed edge from B1 to B2 if

- there is a jump from the last statement of B1 the first statement of B2, or
- B2 follows B1 in the program, and B1 does not end in an unconditional jump.

Flow Graphs for 3-address Programs

Example



Next Use Information

The code generator collects next use information to help decide how to use registers and temporary storage. The general strategy is to keep a name in storage only if it will be used again.

Next use information is collected by scanning the basic blocks backwards, and recording for each name in a block whether the name has a next use in the block, and, if not, whether it is live on exit from the block.

A Simple Code Generator – Aho, Sethi and Ullman

Input – a sequence of 3-address statements

Output – target code

Strategy – consider each of the 3-address statements in turn, "remembering" if any of the operands are currently in registers, and taking advantage of that fact if possible.

Computed results are left in registers as long as possible, and are stored only when the register is needed for another computation, or a procedure call, jump or labelled statement is encountered.

A Simple Code Generator

Important Data Structures

Register Descriptor – current contents of each register.

Address Descriptor – where current value of a name can be found at run time; e.g. in a register, on the stack, in memory.

Important Function

get reg – a procedure that returns the address of a location L in which to store the result of a 3-address assignment. Usually L will be a register, but it could be a memory location.

Code Generation Algorithm

For 3-address statements of the form `x := y op z`

- Call 'get reg' to determine the location L where the result of y op z will be stored.
- Look up y', the current location of y in the address descriptor; if y is both in memory and in a register, choose the register. Unless the value of y is already in L, generate `MOV y', L`
- Determine z', the current location of z in and generate `OP z', L`
- Update the address descriptor to indicate that L is the location of the current value of x. If L is a register, update its register descriptor to indicate that it holds the value of x, and remove x from all other register descriptors.
- If the current values of y (or z) has no next uses, and is not live on exit from the current basic block, and is in a register, then alter the register descriptor to free the register.

Code Generation Algorithm – continued

Code generation is done in a similar way for `x := op y`

For a statement like `x := y`

- If y is in a register, modify the address and register descriptors to say that x is now there. If y has no next use, modify the register descriptors to say that y is no longer there.
- If y is in memory, and x has no next use in the block, generate `MOV y, x`
- If y is in memory, and x has a next use in the block, use 'getreg' to find a register in which to load y, and make that register the location of x.

Example

Generating code for

```
d:= (a-b) + (a-c) + (a-c)
```

3-address code	target code	register descriptor	address descriptor
		all registers empty	
t := a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u := a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v := t + u	ADD R1, R0	R0 contains v R1 contains u	v in R0 u in R1
d := v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 and in memory

Optimisation

Optimisation means using algorithms and heuristics with the aim of improving the code generated by the compiler.

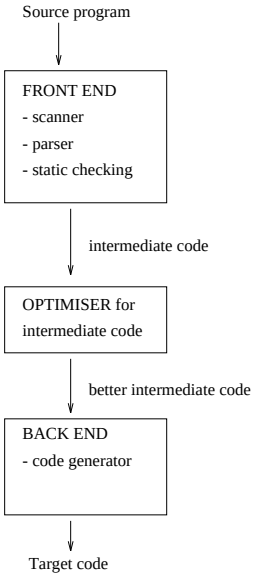
An immense number of these are known.

Some are used in almost all production compilers.

Others are only used in special *optimising compilers*.

The term "optimising compiler" is not strictly accurate. Optimisation includes some undecidable problems, and some whose solution is prohibitively expensive.

Structure of an Optimising Compiler



Criteria by which Optimisations are Judged (Fischer, LeBlanc)

Optimisations are judged by their safety and profitability.

For example suppose the loop

```
while J < I loop
    A(J) := 10/I;
    J := J+2;
end loop;
```

is transformed to

```
t1 := 10/I;
while J < I loop
    A(J) := t1;
    J := J+2;
end loop;
```

What if I=0 or J≥I initially?

Classification of Optimisations

1. Source Language Optimisations are

- performed by semantic routines,
- language specific and
- machine independent

Source Language Optimisations

Example: Loop Unscrolling

```
for I in 1..10 loop
    A(I) := 2*I;
end loop;
```

is changed to

```
A(1) := 2;
A(2) := 4;
...
A(10):= 20;
```


Classification of Optimisations

2. Target Code Optimisations

- exploit target machine architecture and
- are source language independent.

Example:

```
MPY 2,Y
```

is changed to

```
SHIFTLEFT y
```

Classification of Optimisations

3. Intermediate Representation Optimisations

- depend solely on the intermediate representation,
- can be shared by many compilers for different source languages or target machines or both
- entail processing flow graphs of basic blocks

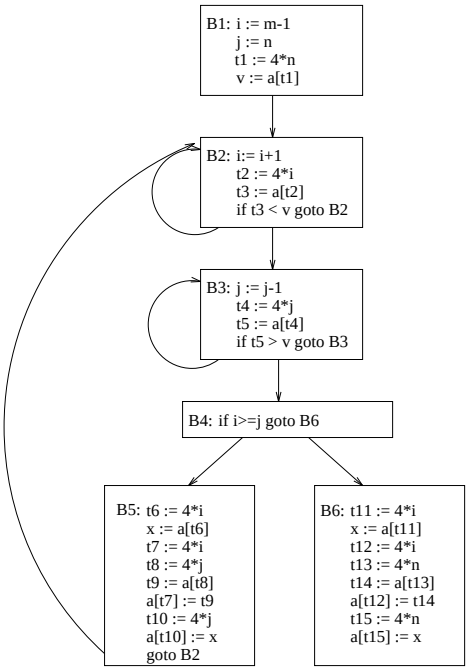
Flow Graph for a quicksort program (Aho, Sethi and Ullman)

Consider the following flow graph for a quicksort program.

The graph is from Aho, Sethi and Ullman, page 591, Figure 10.5; the optimisation examples that follow have all been taken from the same source.

The code assumes that each element of the array a takes 4 bytes, so moving from one element of the array to the next means adding 4 to the array index.

Flow Graph for a quicksort program



Some Useful Optimisations

- elimination of common subexpressions
- copy propagation
- dead code elimination
- loop optimisations
  - code motion
  - induction variable elimination
  - strength reduction

Local elimination of common subexpressions

Consider block B5 from the quicksort program. We can avoid recomputing 4*i and 4*j in this block.

Before

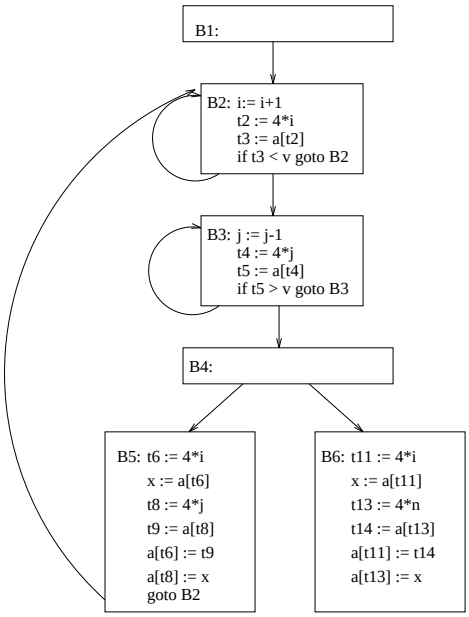
```
B5: t6 := 4*i
    x  := a[t6]
    t7 := 4*i
    t8 := 4*j
    t9 := a[t8]
    a[t7] := t9
    t10 := 4*j
    a[t10] := x
    goto B2
```

After

```
B5: t6 := 4*i
    x  := a[t6]
    t8 := 4*j
    t9 := a[t8]
    a[t6] := t9
    a[t8] := x
    goto B2
```

Similarly, we can avoid recomputing 4*i and 4*n in B6.

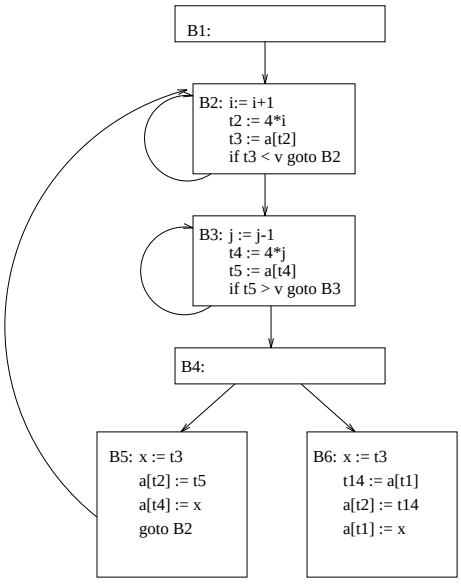
Quicksort flow graph after local elimination of common subexpressions in blocks B5 and B6



Global elimination of common subexpressions

- 4*i is assigned to t2 in B2; this value is still valid in B5 and B6.
- 4*j is assigned to t4 in B3, and is still valid in B5.
- 4*n is assigned to t1 in B1, and is still valid in B6.
- a[t2] is assigned to t3 in B2, and is still valid in B5 and B6.
- a[t4] is assigned to t5 in B3, and is still valid in B5.

Quicksort flow graph after global elimination of common subexpressions



Copy propagation

Following a copy statement `f := g` use `g` in place of `f` wherever possible.

Consider block B5 after global elimination of common subexpressions

Before

```
B5: x := t3
    a[t2] := t5
    a[t4] := x
    goto B2
```

After

```
B5: x := t3
    a[t2] := t5
    a[t4] := t3
    goto B2
```

So what?

Dead code elimination

A variable is live at a point in a program if its value is subsequently used; otherwise it is dead.

A piece of code is live if the value(s) it computes is (are) used subsequently; otherwise it is dead.

Copy propagation often turns the copy statement into dead code, which can be eliminated.

Block B5 again

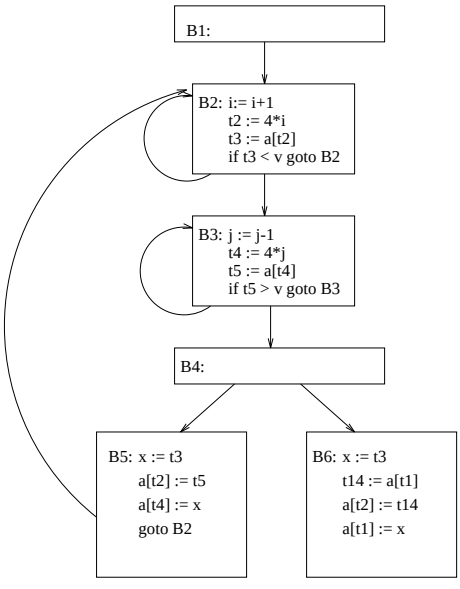
Before

```
B5: x := t3
    a[t2] := t5
    a[t4] := t3
    goto B2
```

After

```
B5: a[t2] := t5
    a[t4] := t3
    goto B2
```


Quicksort flow graph after copy propogation and dead code elimination in B5 and B6



Code Motion

Move code out of a loop to a point before the loop; this avoids recomputing values.

This doesn't happen in the quicksort example, so here is a different example.

Before

```
Start: m := 10
       n := 50
Loop:  a[n] := m*4
       b[n] := m+15
       n := n-1
       if n >= 1 goto Loop
```

After

```
Start: m := 10
       n := 50
       t1 := m*4
       t2 := m+15
Loop:  a[n] := t1
       b[n] := t2
       n := n-1
       if n >= 1 goto Loop
```

Note: this is not part of the quicksort example!

Induction Variable Elimination

In the loop round B2 in the quicksort flow graph, every time `i` is incremented

```
t2 := 4*i
```

is recalculated. This means that the relationship

```
t2 = 4*i
```

is maintained.

`i` and `t2` are called induction variables

It is often possible to eliminate all but one of the induction variables in a loop.

However, before this can be done in the quicksort example, some further manipulation is needed.

Strength Reduction

Replace slower operations (like multiplication) by faster ones (like addition) in a loop.

In block B2, provided

```
t2 = 4*i
```

the statements

```
i := i+1
t2 := 4*i
```

can be replaced by

```
i := i+1
t2 := t2+4
```

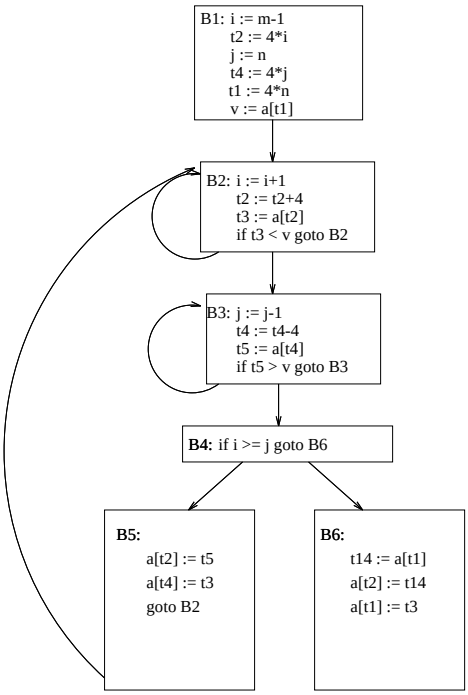
It remains to establish

```
t2 = 4*i
```

at the point of entry to B2. This is done by modifying B1, where i is initialised.

A similar transformation can be carried out in B3.

Strength Reduction in blocks B2 and B3



Back to induction variable elimination

Consider `i` and `j`.

After strength reduction in B2 and B3, `i` and `j` are only used in the test in B4.

Because

```
t2 = 4*i
```

and

```
t4 = 4*j
```

this test can be replaced by

```
t2 >= t4
```

Also, the statements

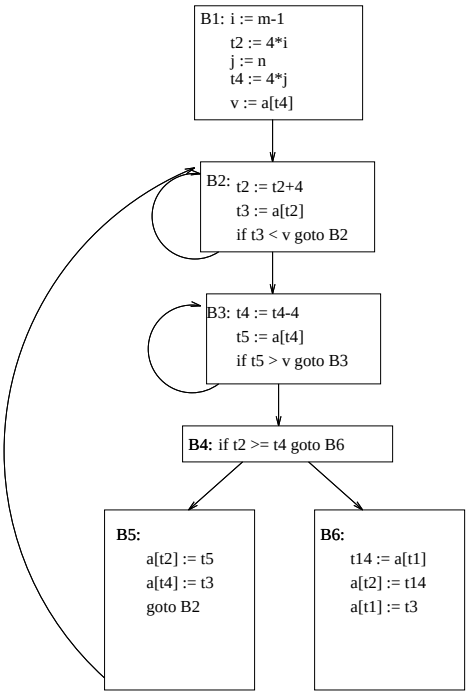
```
i := i+1
```

in B2 and

```
j := j-1
```

in B3 are now dead code, and can be eliminated.

The final flow graph



Compilation Systems - Summary

- Structure of a compiler
- Lexical analysis (scanning)
- Syntactic analysis (parsing)
- Semantic analysis (context sensitive checking)
- Code Generation
- Optimisation