

Department of Computer Science
Australian National University

COMP3610
Principles of Programming Languages

An Introduction to Parsing and Translation

Clem Baker-Finch
August 8, 2007

Contents

1	Overview	1
2	Basic Components of a Compiler	1
3	Parsing	4
3.1	Ambiguity	7
3.2	Left Recursive Productions	8
3.3	Left Factoring Productions	9
3.4	Director Symbols	10
3.4.1	Deciding which non-terminals derive ϵ	11
3.4.2	Finding <i>starter symbols</i> for each non-terminal	12
3.4.3	Finding <i>follower symbols</i> for each non-terminal	14
4	Haskell Parser and Interpreter for <i>Imp</i>	17

1 Overview

Basic concepts and terminology. Introduction to the techniques of translation of programming languages and the structure of compilers.

Terminology

A **translator** is a program that accepts as input the representation of a program in some notation (the **source** language). It produces as output a representation of the same program in some other notation (the **target** language). Since the translator is itself a program, it is written in some notation as well (the **host** language).

A **compiler** is a translator where the source language is usually a high-level problem-oriented programming language and the target is a low-level language such as machine code or assembler.

This is not a precise definition. Compare the usual idea of an *assembler* where the source language is (more or less) a symbolic form of the target language (machine code).

2 Basic Components of a Compiler

Lexical Analysis — *Scanning*

Input: text of source program.

Output: a sequence of tokens.

The lexical analyser takes the source program text (a sequence of characters) as input and breaks it into basic units, such as keywords, operators, punctuation and identifiers. We usually call these units *tokens*.

Syntactic Analysis — *Parsing*

Input: a sequence of tokens.

Output: a parse tree for the program.

The syntax analyser (the *parser*) analyses the sequence of tokens to retrieve the “structure” of the program. For example, the body of a loop, the expression in an assignment, etc.

Semantic Analysis

Input: a parse tree.

Output: a flattened intermediate form of the program.

The semantic analysis phase traverses the parse tree and produces a sequence of instructions in an assembly-like language. It is *flat* in the sense that all the nested structures (loops etc.) from the source program have been eliminated (replace by explicit tests and branches).

Code Generation

Input: a flattened intermediate form.

Output: assembly language, relocatable binary or some such machine code.

Produce code (either assembly or binary) taking account of the machine model of the source language and the actual target machine.

Assembler, Loader, . . .

If a compiler produces an assembly language program, then these steps are required. However they are not really part of the compiler proper.

Error Handler

Any realistic, useful compiler must report helpful error messages. Errors can arise in the lexical analysis phase (e.g. illegal characters), or in the syntactic analysis phase (e.g. missing semi-colons), or in the semantic analysis phase (e.g. undeclared identifiers, function headers with no bodies etc.)

Symbol Tables

The syntactic and semantic analysis phases both require information about the *identifiers* or *names* introduced in the source program. The kind of information needed may be whether it is the name of a function, parameter, variable, constant etc. For variables, we may want to their type, their size and so on. For constants, we will want to know their values. For functions we may want to know the number of parameters and their types and so on.

ll this information is maintained in data structures called symbol tables. They also need to keep track of the *scope* of names (i.e. nested declarations).

Example

Imagine a program in some *source language*:

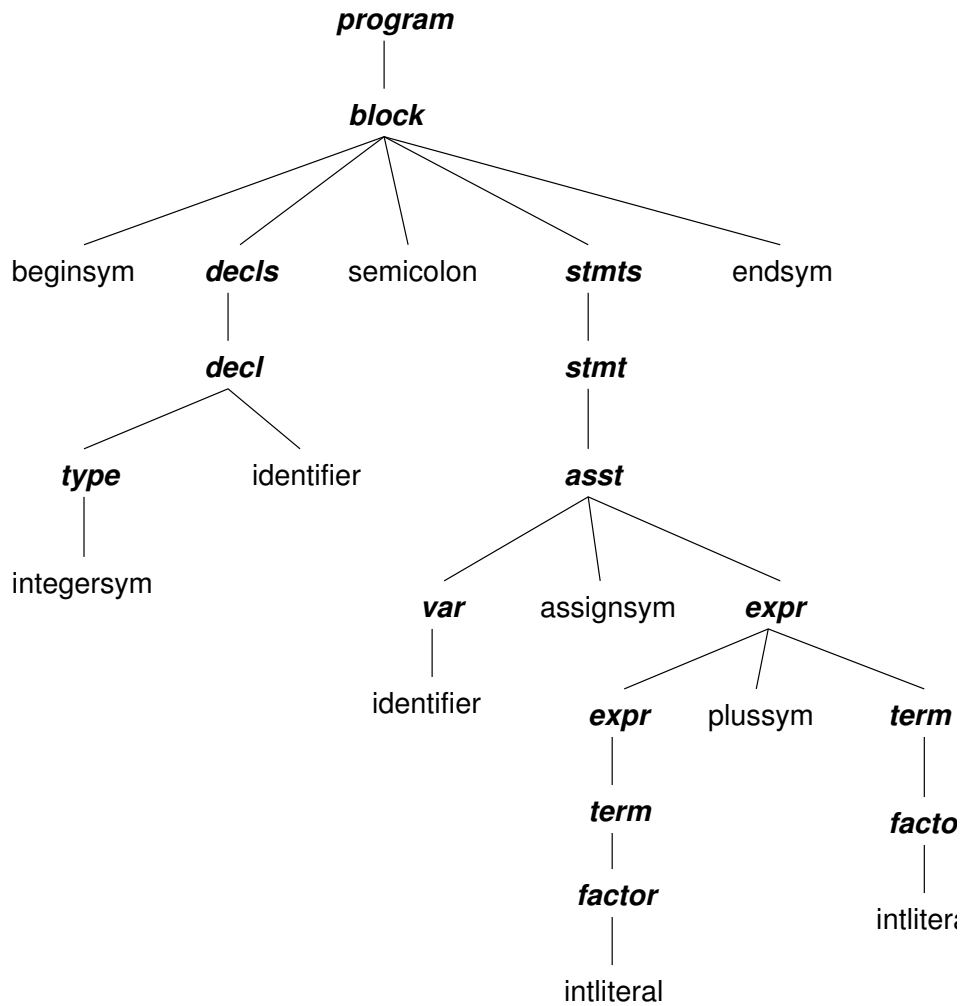
```
begin
    integer i;
    i := 3+1
end
```

The lexer (lexical analyser) produces the following sequence of tokens:

```
beginsym
integersym
identifier (i)
semicolon
identifier (i)
assignsym (:=)
```

intliteral (3)
plussym
intliteral (1)
endsym

The parser (syntactic analyser) produces the following parse tree:



The semantic analysis phase produces the following intermediate code:

```

DATA L1 INT
LAB L2
LOAD 3
LOAD 1
PLUS
STORE L1
START L2

```

Notice that the essence of the semantic analysis phase is to *flatten* the parse tree in a way that reflects the execution of the program.

Finally the code generator produces the following assembly code:

```

L1      .WORD
L2      MOV      =3,  R0
        ADD      =1,  R0
        MOV      R0,  L1
        .END     L2

```

Notes

- Not all of the phases described are necessarily distinct or clearly defined.
- Not all intermediate forms are necessarily present as data structures in the compiler.
- Usually the phases are *interleaved*. That is, once enough tokens have been recognised to start building the parse tree, do so. Once enough of the parse tree has been built to start generating intermediate code, do so.
- As the process progresses there is *increasing machine dependency*. Hence the later phases are more *ad hoc*, but the lexical and syntactic analysis phases are based on well-known and understood theory. (One of the great successes of Computer Science.)
- Optimisations may occur at several levels, but mostly in the semantic analysis and code generation phases.

3 Parsing

Parsing is essentially the process of determining if a string of tokens can be generated by a grammar. That is, build a parse tree by filling in the tree between the root (i.e. the start symbol) and the leaves (i.e. the string of tokens). Compilers are generally structured around a parser.

Top-down parsing: construction of the parse tree starts at the root and proceeds down to the leaves (depth-first).

Bottom-up parsing: construction of the parse tree starts at the leaves and proceeds up to the root (depth-first).

Top-down is usually the choice for hand-built compilers. Bottom-up copes with a wider range of grammars and is almost always the method used by compiler generators.

Top-down example:

Consider the following grammar:

```

type  → simple | * id | array [ simple ] of type
simple → integer | char | num elipsis num

```

(Assume we have a lexer to recognise all the tokens.) The process of top-down parsing is:

- **At a node with non-terminal N , select a production for N**

- Construct children at that node for the symbols of the rhs of that production
- Proceed to the next node (depth-first)

For example, to parse `array [num elipsis num] of integer` start with the root (i.e. *type*). Add its children and continue in depth-first order. (Exercise.)

Note that for this grammar the *next token in the stream* always uniquely determines which production to select.

Recursive Descent parsers are top-down parsers that consist of a collection of functions, one associated with each non-terminal of the grammar.

Within each of those functions, depending on the lookahead symbol (i.e. the next token in the input stream), either a terminal symbol is checked and skipped or a (non-terminal) function is called, in sequence according to the rhs of each production.

A Haskell implementation of such a parser is the script `ParseTypeDefs.lhs` listed below and also available on the course web site. This first example was chosen because it works without modification. In general there are several problems to be overcome by modifying the grammar before a top-down parser can be constructed:

- The grammar may be *ambiguous*
- The productions may be *left-recursive*
- A *single lookahead symbol* may not be sufficient to predict which production will guide the next step in the parse

`ParseTypeDefs.hs`

```

module ParseTypeDefs where

import Char

-- The grammar is suitable for top-down predictive parsing without
-- modification:

-- <type> ::= <simple> | * ident | array [ <simple> ] of <type>
-- <simple> ::= int | char | num .. num

-- First the abstract syntax, the target of the parser:

type Name = String

data TypeDef = Simple Simple | Ptr Name | Arr Simple TypeDef
  deriving Show

data Simple = IntType | CharType | Range Int Int
  deriving Show

-- The scanner is straightforward:

```

```

data Token = STAR | ID Name
           | LBRAK | RBRAK
           | ARRAY | OF
           | INTSYM | CHARSYM
           | NUM Int | ELIPSIS deriving Eq

scan :: String -> [Token]

scan [] = []
scan ('*':cs) = STAR : scan cs
scan ('[':cs) = LBRAK : scan cs
scan (']':cs) = RBRAK : scan cs
scan ('.':'.':cs) = ELIPSIS : scan cs
scan input@(c:cs)
  | isSpace c = scan cs
  | isAlpha c = checkResWord word : scan afterWord
  | isDigit c = NUM (read num) : scan afterNum
  | c == '-' = NUM (read num) : scan afterNum
  | otherwise = error (c:" : illegal character.")
  where
    (word, afterWord) = span isAlphaNum input
    (num, afterNum) = span isDigit input

checkResWord :: String -> Token

checkResWord "int" = INTSYM
checkResWord "char" = CHARSYM
checkResWord "array" = ARRAY
checkResWord "of" = OF
checkResWord other = ID other

-- The parser takes a sequence of tokens and constructs a parse tree
-- (i.e. a value of TypeDef). It must also return the sequence of
-- remaining tokens with which to continue the parse. There will also
-- be a parser corresponding to the syntactic category Simple.

parseTypeDef :: [Token] -> (TypeDef, [Token])

parseTypeDef (STAR:ID name:toks) = (Ptr name, toks)
parseTypeDef (ARRAY:LBRAK:toks) = (Arr ixType elType, toks4)
  where (ixType, toks1) = parseSimple toks
        toks2 = check RBRAK toks1
        toks3 = check OF toks2
        (elType, toks4) = parseTypeDef toks3
parseTypeDef toks = (Simple sType, toks')
  where (sType, toks') = parseSimple toks

parseSimple :: [Token] -> (Simple, [Token])

parseSimple (INTSYM:toks) = (IntType, toks)
parseSimple (CHARSYM:toks) = (CharType, toks)
parseSimple (NUM min : ELIPSIS : NUM max : toks) = (Range min max, toks)
parseSimple _ = error "Unexpected symbol."

```

```

-- Check that the next token is the one expected:

check :: Token -> [Token] -> [Token]

check tok [] = error "Unexpected end of input."
check tok (t:ts)
  | tok == t = ts
  | otherwise = error "Unexpected symbol."

-- To parse an entire type definition, simply require all tokens to be
-- consumed.

parse :: String -> TypeDef
parse input = typeDef
  where (typeDef, []) = parseTypeDef $ scan input

```

3.1 Ambiguity

Consider the following two grammars for the same language.

Grammar 1:

$$\begin{aligned}
 E &\rightarrow T \mid E + T \mid E - T \\
 T &\rightarrow F \mid T * F \\
 F &\rightarrow \text{id} \mid (E)
 \end{aligned}$$

Grammar 2:

$$E \rightarrow \text{id} \mid E + E \mid E - E \mid E * E \mid (E)$$

Definition 3.1

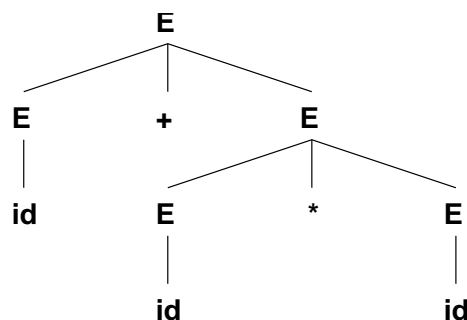
A grammar is *ambiguous* if some sentence has more than one parse tree.

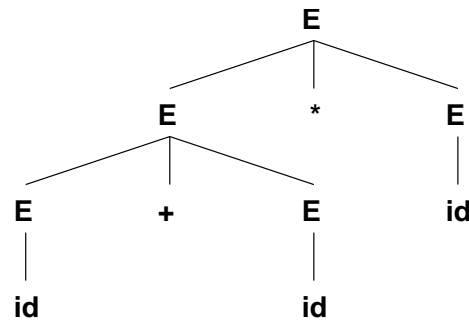
The problem with ambiguous grammars is that they do not suggest deterministic parsing algorithms. A language may have both an ambiguous grammar and an unambiguous grammar (for example, grammars 1 and 2 above). Ambiguity is an undecidable property.

For the sentence

`id + id * id`

both the following parse trees can be constructed from Grammar 2:





In comparison, for grammar 1 there is only *one parse tree* for any sentence. There is another important advantage of grammar 1:

The parse tree reflects the priority and left-associativity of the operators.

This is very important because it simplifies further processing and translation.

Exercise:

Consider how you would write an evaluator for expression parse trees. What would be the consequences of working with the following grammar?

$$E \rightarrow T \mid T + E \mid T - E$$

$$T \rightarrow F \mid F * T$$

$$F \rightarrow \mathbf{id} \mid (E)$$

3.2 Left Recursive Productions

Suppose we were trying to write a parser based on the usual simple expression grammar:

$$E \rightarrow T \mid E + T \mid E - T$$

$$T \rightarrow F \mid T * F$$

$$F \rightarrow \mathbf{id} \mid (E)$$

The function for parsing factors would be straightforward, say:

```

parseF :: [Token] -> (Factor, [Token])
parseF (ID name : toks) = ...
parseF (LPAREN  : toks) = ...
  
```

But what about $T \rightarrow F \mid T * F$?

A lookahead of **id** (for example) is *compatible with both productions*.

Definition 3.2 (Left Recursion)

A grammar is *left-recursive* if there is some non-terminal A such that $A \xRightarrow{\pm} A\alpha$.

If the grammar includes $A \rightarrow \beta \mid \gamma$ and $\gamma \xRightarrow{*} A\alpha$, then if the next token can start β it is impossible to tell whether to get there by production $A \Rightarrow \beta$ or by the derivation $A \Rightarrow \gamma \xRightarrow{*} A\alpha \Rightarrow \beta\alpha$.

Note that in general there can also be *mutual* left recursion, i.e. via “cycles.” Clearly, whenever we have left-recursion it is impossible to predict (without unlimited lookahead) which production to select.

There is a general algorithm for eliminating left recursion. The following transformation is sufficient for direct left recursion:

Replace all sets of productions like:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \dots \mid \beta_1 \mid \beta_2 \dots$$

with:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \dots$$

$$A' \rightarrow \epsilon \mid \alpha_1 A' \mid \alpha_2 A' \dots$$

where ϵ represents an empty right hand side.

For the expression grammar, that gives:

$$E \rightarrow TE'$$

$$E' \rightarrow \epsilon \mid +TE' \mid -TE'$$

$$T \rightarrow FT'$$

$$T' \rightarrow \epsilon \mid *FT'$$

$$F \rightarrow \mathbf{id} \mid (E)$$

See the textbook for an algorithm that handles mutual recursion.

3.3 Left Factoring Productions

Suppose we have a grammar of simple boolean expressions, built on top of the arithmetic expressions above:

$$bexp \rightarrow bexp \mathbf{and} bterm \mid bexp \mathbf{or} bterm \mid bterm$$

$$bterm \rightarrow \mathbf{not} bterm \mid exp = exp \mid exp \leq exp$$

Eliminating left recursion gives:

$$bexp \rightarrow bterm bexpOpt$$

$$bexpOpt \rightarrow \mathbf{and} bterm bexpOpt \mid \mathbf{or} bterm bexpOpt \mid \epsilon$$

$$bterm \rightarrow \mathbf{not} bterm \mid exp = exp \mid exp \leq exp$$

A problem remains. It is not possible to choose between the two productions:

$$bterm \rightarrow exp = exp \mid exp \leq exp$$

based on a single lookahead token.

The solution is to *defer* the decision by first parsing an *exp*, and then looking at the relational operator to decide which production is the correct choice. This can be expressed in the grammar by *left-factoring* the offending productions:

$$\begin{aligned} bterm &\rightarrow exp \text{ relSection } | \dots \\ relSection &\rightarrow = exp | <= exp \end{aligned}$$

In general, the algorithm to left factor is:

$$\begin{aligned} \text{Replace } &A \rightarrow \alpha\beta_1 | \alpha\beta_2 \dots \\ \text{by } &A \rightarrow \alpha A' \\ \text{and } &A' \rightarrow \beta_1 | \beta_2 \dots \end{aligned}$$

(Obviously we must eliminate left-recursion first.) No left factoring is needed for the modified expression grammar above.

3.4 Director Symbols

Provided the original grammar is not ambiguous, eliminating left-recursion and left-factoring will produce a grammar which can guide a top-down parser. (The resultant grammar may be less natural than the original.)

For example, the simple expression grammar transforms to:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow \epsilon | +TE' | -TE' \\ T &\rightarrow FT' \\ T' &\rightarrow \epsilon | *FT' \\ F &\rightarrow \mathbf{id} | (E) \end{aligned}$$

The productions are chosen as follows (\dagger represents end of input):

$$\begin{array}{lll} E \rightarrow TE' & \text{on lookahead} & \mathbf{id}, (\\ E' \rightarrow \epsilon & \text{on lookahead} &), \dagger \\ E' \rightarrow +TE' & \text{on lookahead} & + \\ T \rightarrow FT' & \text{on lookahead} & \mathbf{id}, (\\ T' \rightarrow \epsilon & \text{on lookahead} & +,), \dagger \\ T' \rightarrow *FT' & \text{on lookahead} & * \\ F \rightarrow \mathbf{id} & \text{on lookahead} & \mathbf{id} \\ F \rightarrow (E) & \text{on lookahead} & (\end{array}$$

Exercise 3.3

Use this scheme to parse some sample strings such as $(\mathbf{id}+\mathbf{id})*\mathbf{id}$. Include some illegal strings.

Definition 3.4

The lookahead symbols that guide the choice of production are called the **director** symbols of each production.

How do we *find* the director symbols for a grammar?

- For productions whose right hand sides begin with a terminal symbol the director symbol is that symbol. For example the director symbol of $E' \rightarrow +TE'$ must be $+$.
- What about $E \rightarrow TE'$? The director symbols are all the terminal symbols that can **start** a string derivable from T .
- What about $E' \rightarrow \epsilon$? The director symbols must be any terminal symbol that can **follow** E' in a sentential form. (In fact, we need to consider *derivations* $A \xRightarrow{*} \epsilon$ as well as direct productions.)

Therefore, to find the director symbols we need to know:

1. Which non-terminals can derive the empty string?
2. Which terminal symbols can **start** strings derivable from each non-terminal?
3. Which terminal symbols can **follow** each non-terminal identified in (1)?

These three pieces of information are not independent: to find **follow** we need **start**, and both rely on information about empty derivations.

3.4.1 Deciding which non-terminals derive ϵ

Set up a table with one entry per non-terminal and initialise each to *undecided*.

```

while there are production left to consider
  if there exists a production for  $A$  whose rhs is  $\epsilon$ 
    or is only non-terminals, all marked  $\epsilon$  in the table then
      mark  $A$  with  $\epsilon$  in the table;
      remove all  $A$ -productions;
    elsif the rhs's of all productions for  $A$  contain at least one terminal
      or a non-terminal marked not- $\epsilon$  in the table then
        mark  $A$  with not- $\epsilon$  in the table;
        remove all  $A$ -productions;
  endwhile

```

In other words, look for non-terminals that certainly do or certainly do not derive ϵ , then propagate that information.

Example 3.5

$$\begin{aligned}
 E &\rightarrow TE' \\
 E' &\rightarrow \epsilon \mid +TE' \\
 T &\rightarrow FT' \\
 T' &\rightarrow \epsilon \mid *FT' \\
 F &\rightarrow \mathbf{id} \mid (E)
 \end{aligned}$$

E	E'	T	T'	F

Following the algorithm shows only E' and T' able to derive ϵ .

Immediate entries:

E	E'	T	T'	F
	ϵ		ϵ	<i>not</i> ϵ

Transitive entries:

E	E'	T	T'	F
<i>not</i> ϵ	ϵ	<i>not</i> ϵ	ϵ	<i>not</i> ϵ

3.4.2 Finding *starter symbols* for each non-terminal

The basic idea is as follows:

- If there is a production $A \rightarrow a \dots$ then a *can start* A .
- If there is a production $A \rightarrow B \dots$ then *any starter of* B *can start* A .

There is a small complication caused by empty productions:

- If there is a production $A \rightarrow \alpha a \dots$ and $\alpha \xRightarrow{*} \epsilon$ then a can start A .
- If there is a production $A \rightarrow \alpha B \dots$ and $\alpha \xRightarrow{*} \epsilon$ then any starter of B can also start A .

The “starter” relation is *transitive*. What we require is an algorithm to calculate the *transitive closure*.

Example 3.6

From the productions:

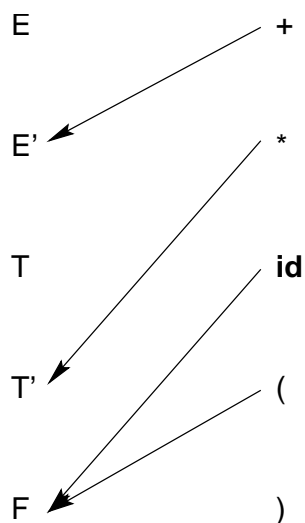
$$E' \rightarrow +TE'$$

$$T' \rightarrow *FT'$$

$$F \rightarrow \mathbf{id}$$

$$F \rightarrow (E)$$

we get the following terminal starter information, represented as a graph:

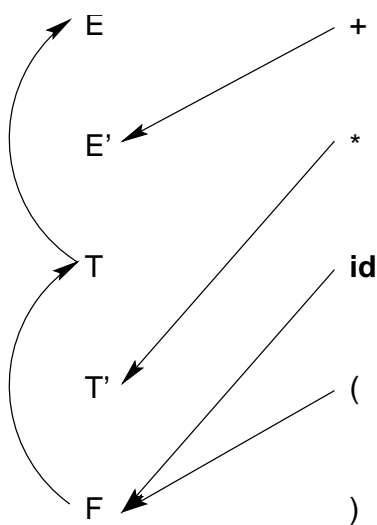


where edges represent the “starts” relation. The transitivity comes from the productions:

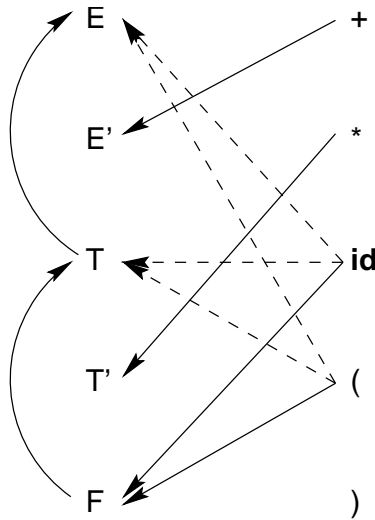
$$E \rightarrow TE'$$

$$T \rightarrow FT'$$

which we add to the graph:



Taking the transitive closure means adding edges between any nodes connected by a path. This gives:



So the starter symbols are:

- E : id, (
- E' : +
- T : id, (
- T' : *
- F : id, (

3.4.3 Finding *follower symbols* for each non-terminal

The basic idea is as follows:

- If there is a production $A \rightarrow \dots Ba \dots$ then a *can follow* B .
- If there is a production $A \rightarrow \dots BC \dots$ then *any starter of C can follow B*.
- If there is a production $A \rightarrow \dots B$ then *any follower of A can also follow B*.

The same complication arises again. Suppose $\alpha \xRightarrow{*} \epsilon$. Then:

- In the first case, include $A \rightarrow \dots B\alpha a \dots$
- In the second case, include $A \rightarrow \dots B\alpha C \dots$
- In the third case, include $A \rightarrow \dots B\alpha$

Again, the “follower” relation is *transitive* and we require the *transitive closure*.

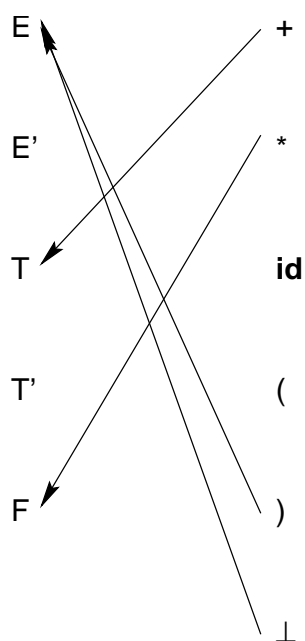
Example 3.7

From $F \rightarrow (E)$ we know that $)$ can follow E .

From $E \rightarrow TE'$ we know that the starters of E' can follow T , so $+$ can follow T .

Similarly, from $T \rightarrow FT'$ we know that $*$ can follow F .

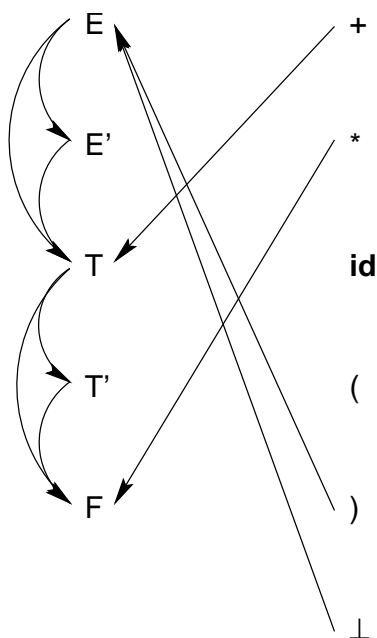
(As a matter of course, end-of-input can follow the start symbol.) This can be represented as the following graph:



From $E \rightarrow TE'$ we know that whatever can follow E can also follow E' .

Since $E' \xRightarrow{*} \epsilon$, we know that the followers of E can also follow T .

Similarly for $E' \rightarrow +TE'$, $T \rightarrow FT'$ and $T' \rightarrow *FT'$, noting that $E' \xRightarrow{*} \epsilon$ and $T' \xRightarrow{*} \epsilon$, giving the graph:



Taking the transitive closure of this graph/relation gives the follower symbols:

$$\begin{aligned}
 E &:), \vdash \\
 E' &:), \vdash \\
 T &: +,), \vdash \\
 T' &: +,), \vdash \\
 F &: *, +,), \vdash
 \end{aligned}$$

Note that we are only interested in the followers of E' and T' (since they are the only non-terminals that can derive the empty string) but they cannot be calculated in isolation — the followers of T' include the followers of E , E' and T .

All the procedures above (left recursion elimination, left-factoring, finding director symbols) have been applied to a grammar for a simple language *Imp* which will be discussed in the *Semantics* section of the course. The scripts for the resultant Haskell parser are available at the unit web site.

The original (ambiguous) grammar and the grammar resulting from the modifications follow. This information, with more discussion (and the director symbols) appear in the script `Parser.lhs`.

Original grammar for *Imp*

$$\begin{aligned}
 expr &\rightarrow expr + term \mid expr - term \mid term \\
 term &\rightarrow term * factor \mid factor \\
 factor &\rightarrow \mathbf{num} \mid \mathbf{id} \mid (expr) \\
 bexpr &\rightarrow bexpr \mathbf{and} bterm \mid bexpr \mathbf{or} bterm \mid bterm \\
 bterm &\rightarrow \mathbf{not} bterm \mid \mathbf{true} \mid \mathbf{false} \\
 &\quad \mid expr = expr \mid expr <= expr \\
 com &\rightarrow \mathbf{id} := expr \mid \mathbf{if} bexpr \mathbf{then} com \mathbf{else} com \\
 &\quad \mid \mathbf{while} bexpr \mathbf{do} com \mid \{ comSeq \} \\
 comSeq &\rightarrow com \mid comSeq ; comSeq \mid \epsilon \\
 program &\rightarrow comSeq
 \end{aligned}$$

Deal with the ambiguity first:

$$comSeq \rightarrow com \mid \epsilon \mid com ; comSeq \mid ; comSeq$$

Modified grammar for *Imp*

```

expr      → term expOpt
expOpt    → + term expOpt | - term expOpt |  $\epsilon$ 
term      → factor termOpt
termOpt   → * factor termOpt |  $\epsilon$ 
factor    → num | id | (expr)

bexpr     → bterm bexpOpt
bexpOpt   → and bterm bexpOpt | or bterm bexpOpt |  $\epsilon$ 
bterm     → not bterm | true | false | expr relSection
relSection → = expr | <= expr

com       → id := expr | if bexpr then com else com
           | while bexpr do com | { comSeq }
comSeq    → com seqOpt | ; comSeq |  $\epsilon$ 
seqOpt    → ; comSeq |  $\epsilon$ 

program   → comSeq

```

4 Haskell Parser and Interpreter for *Imp*

AbSyn.hs

```

module AbsSyn where

type Name = String

data Aexp = Num Int
          | Var Name
          | Aexp :+: Aexp
          | Aexp :-: Aexp
          | Aexp *: Aexp
          deriving Show

data Bexp = TrueLit
          | FalseLit
          | Aexp :=: Aexp
          | Aexp <=: Aexp
          | Not Bexp
          | Bexp 'And' Bexp
          | Bexp 'Or' Bexp
          deriving Show

data Com = Name := Aexp
         | Skip
         | Com :~: Com
         | If Bexp Com Com
         | While Bexp Com
         deriving Show

```

Scanner.hs

```

module Scanner (Token(..), scan, unscan) where

import Data.Char

data Token = ID String
           | NUM Int
           | PLUS | MINUS | TIMES
           | TRUE | FALSE | EQUAL | LEQ
           | NOT | AND | OR
           | ASSIGN | LBRACE | RBRACE
           | IF | THEN | ELSE | WHILE | DO
           | SEMICOLON | LPAREN | RPAREN deriving (Eq, Show)

-- The scan function returns a list of tokens.

scan :: String -> [Token]

scan [] = []
scan ('{' ':' '-' ':cs) = scan (skipComment cs 1)
scan ('=' :cs) = EQUAL : scan cs
scan ('+' :cs) = PLUS : scan cs
scan ('-' :cs) = MINUS : scan cs
scan ('*' :cs) = TIMES : scan cs
scan (',' :cs) = SEMICOLON : scan cs
scan ('{' :cs) = LBRACE : scan cs
scan ('}' :cs) = RBRACE : scan cs
scan ('(' :cs) = LPAREN : scan cs
scan (')' :cs) = RPAREN : scan cs
scan ('<' ':' '=' :cs) = LEQ : scan cs
scan (':' ':' '=' :cs) = ASSIGN : scan cs
scan input@(c:cs)
  | isSpace c = scan cs
  | isAlpha c = checkResWord word : scan afterWord
  | isDigit c = NUM (read num) : scan afterNum
  | otherwise = error (c:" : illegal character.")
  where
    (word, afterWord) = span isAlphaNum input
    (num, afterNum) = span isDigit input

-- Skip comments delimited by {- ... -}.
-- Handles nested comments. The 2nd argument is the nesting depth.

skipComment :: String -> Int -> String

skipComment [] depth = error "End of input while scanning comment."
skipComment [_] depth = error "End of input while scanning comment."
skipComment ('-' ':' '-' ':'cs) depth
  | depth == 1 = cs
  | otherwise = skipComment cs (depth-1)
skipComment ('{' ':' '-' ':'cs) depth
  = skipComment cs (depth+1)
skipComment (_:cs) depth = skipComment cs depth

-- Distinguish reserved words from identifiers.

```

```

-- Case sensitive for simplicity.

checkResWord :: String -> Token

checkResWord "true"  = TRUE
checkResWord "false" = FALSE
checkResWord "not"   = NOT
checkResWord "and"   = AND
checkResWord "or"    = OR
checkResWord "if"    = IF
checkResWord "then"  = THEN
checkResWord "else"  = ELSE
checkResWord "while" = WHILE
checkResWord "do"    = DO
checkResWord other   = ID other

-- For better error messages, convert tokens back to strings.
-- Some redundancy, no doubt.

unscan :: Token -> String

unscan (ID x)      = "Identifier " ++ x
unscan (NUM n)     = "Numeral " ++ show n
unscan PLUS        = "+"
unscan MINUS       = "-"
unscan TIMES       = "*"
unscan TRUE        = "true"
unscan FALSE       = "false"
unscan EQUAL       = "="
unscan LEQ         = "<="
unscan NOT         = "not"
unscan AND         = "and"
unscan OR          = "or"
unscan ASSIGN      = ":@"
unscan LBRACE      = "{"
unscan RBRACE      = "}"
unscan IF          = "if"
unscan THEN        = "then"
unscan ELSE        = "else"
unscan WHILE       = "while"
unscan DO          = "do"
unscan SEMICOLON   = ";"
unscan LPAREN      = "("
unscan RPAREN      = ")"

```

Parser.hs

```

module Parser (parseProg) where

import Scanner
import AbsSyn

-- Check that the next token is the one expected:

```

```

check :: Token -> [Token] -> [Token]

check tok [] = error "Unexpected end of input."
check tok (t:ts)
  | tok == t = ts
  | otherwise = error (unscan tok ++ " expected.\n" ++ unscan t ++ " scanned.")
-- Parse an arithmetic expression.
-- Operator precedence is as usual, represented by the following grammar:

-- expr ::= expr + term | expr - term | term
-- term ::= term * factor | factor
-- factor ::= num | ident | (expr)

-- Eliminate left recursion and left to factor give:

-- expr ::= term exprOpt
-- exprOpt ::= + term exprOpt | - term exprOpt | <empty>
-- term ::= factor termOpt
-- termOpt ::= * factor exprOpt | <empty>
-- factor ::= num | ident | (expr)

-- Director symbols:

-- first(expr) = first(term) = first(factor) = {NUM, ID, LPAREN}

-- first(exprOpt) = {PLUS, MINUS}
-- follow(exprOpt) = {RPAREN, ...}

-- For the full grammar developed below, I reckon the complete definition
-- is:
-- follow(exprOpt) = {RPAREN, EQUAL, LEQ} U follow(bexpr) U follow(com)
--                  = {RPAREN, EQUAL, LEQ, AND, OR,
--                    THEN, ELSE, DO, RBRACE, SEMICOLON, EOI}

-- first(term) = first(factor) = {NUM, ID, LPAREN}

-- first(termOpt) = {TIMES}
-- follow(termOpt) = follow(exprOpt) U {PLUS, MINUS}

-- first(factor) = {NUM, ID, LPAREN}

expr :: [Token] -> (Aexp, [Token])

expr = exprOpt . term

exprOpt :: (Aexp, [Token]) -> (Aexp, [Token])

exprOpt (t1, PLUS:toks) = exprOpt (t1 :+: t2, toks')
  where
    (t2, toks') = term toks
exprOpt (t1, MINUS:toks) = exprOpt (t1 :-: t2, toks')
  where
    (t2, toks') = term toks
exprOpt (t, toks) = (t, toks)

```

```

term :: [Token] -> (Aexp, [Token])

term = termOpt . factor

termOpt :: (Aexp, [Token]) -> (Aexp, [Token])

termOpt (f1, TIMES:toks) = termOpt (f1 **: f2, toks')
  where
    (f2, toks') = factor toks
termOpt (f, toks)      = (f, toks)

factor :: [Token] -> (Aexp, [Token])

factor (NUM n:toks) = (Num n, toks)
factor (ID x:toks)  = (Var x, toks)
factor (LPAREN:toks) = (e, toks')
  where
    (e, toks') = expr toks
    toks''      = check RPAREN toks'
factor (t:toks) = error ("Unexpected symbol: " ++ unscan t)
  where
    (e, toks') = expr toks
    toks''      = check RPAREN toks'

-- Parse a boolean expression.

-- bexpr ::= bexpr and bterm | bexpr or bterm | bterm
-- bterm ::= not bterm | true | false | expr = expr | expr <= expr

-- Even though conjunction and disjunction are associative
-- and equiv precedence we follow the same process as above:

-- bexpr ::= bterm bexprOpt
-- bexprOpt ::= and bterm bexprOpt | or bterm bexprOpt | <empty>
-- bterm ::= not bterm | true | false | expr relSection
-- relSection ::= = expr | <= expr

-- Director symbols:

-- first(bexpr) = first(bterm)

-- first(bexprOpt) = {AND, OR}
-- follow(bexprOpt) = follow(bexpr) = {AND, OR, ...}

-- first(bterm) = first(expr) U {NOT, TRUE, FALSE}

-- first(relSection) = {EQUAL, LEQ}

bexpr :: [Token] -> (Bexp, [Token])

bexpr = bexprOpt . bterm

bexprOpt :: (Bexp, [Token]) -> (Bexp, [Token])

```

```

bexprOpt (bt1, AND:toks) = bexprOpt (bt1 'And' bt2, toks')
  where
    (bt2, toks') = bterm toks
bexprOpt (bt1, OR:toks)  = bexprOpt (bt1 'Or'  bt2, toks')
  where
    (bt2, toks') = bterm toks
bexprOpt (bt, toks)     = (bt, toks)

bterm :: [Token] -> (Bexp, [Token])

bterm (NOT:toks) = (Not bt, toks')
  where
    (bt, toks') = bterm toks
bterm (TRUE:toks) = (TrueLit, toks)
bterm (FALSE:toks) = (FalseLit, toks)
bterm toks = relSection (expr toks)

relSection :: (Aexp, [Token]) -> (Bexp, [Token])

relSection (a1, EQUAL:toks) = (a1 :=: a2, toks')
  where
    (a2, toks') = expr toks
relSection (a1, LEQ:toks) = (a1 :<=: a2, toks')
  where
    (a2, toks') = expr toks
relSection _ = error "Relational operator expected."

-- Parse commands.

-- For maximum convenience, allow "empty commands".
-- For example, {}, {c;}, {;c;;} etc.

-- I've eliminated skip from the source language, but it may still arise in
-- the abstract syntax.

-- com ::= x:=e | if bexpr then com else com | while bexpr do com | { comSeq }
-- comSeq ::= com | comSeq ; comSeq | <empty>

-- This production for comSeq reflects the associativity of ';' but leads
-- to an ambiguous grammar. We need to solve that problem first, so we
-- may as well go for right-associativity to avoid left-recursion
-- problems:

-- comSeq ::= com | <empty> | com ; comSeq | ; comSeq

-- Now left-factoring gives:

-- comSeq ::= com seqOpt | ; comSeq | <empty>
-- seqOpt ::= ; comSeq | <empty>

-- Director symbols:

-- first(com) = {ID, IF, WHILE, LBRACE}

```

```

-- first(comSeq) = first(com) U {SEMICOLON}
-- follow(comSeq) = {RBRACE}

-- first(seqOpt) = {SEMICOLON}
-- follow(seqOpt) = {RBRACE}

-- However, since whole programs are defined by the production:

-- program ::= comSeq

-- the follower sets of comSeq and seqOpt also include end of input:

-- follow(comSeq) = {RBRACE, EOI}
-- follow(seqOpt) = {RBRACE, EOI}

-- The different followers will be accounted for in the context
-- of the com and prog parsers.

com :: [Token] -> (Com, [Token])

com (ID x:toks) = (x := a, toks'')
  where
    toks'      = check ASSIGN toks
    (a, toks'') = expr toks'
com (IF:toks)   = (If b c1 c2, toks2)
  where
    (b,toks')   = bexpr toks
    toks''      = check THEN toks'
    (c1,toks1)  = com toks''
    toks1'      = check ELSE toks1
    (c2,toks2)  = com toks1'
com (WHILE:toks) = (While b c, toks''')
  where
    (b, toks') = bexpr toks
    toks''     = check DO toks'
    (c, toks''') = com toks''
com (LBRACE:toks) = (c, toks'')
  where
    (c, toks') = comSeq toks
    toks''     = check RBRACE toks'

comSeq :: [Token] -> (Com, [Token])

comSeq []          = (Skip, [])
comSeq toks@(RBRACE:_) = (Skip, toks)
comSeq (SEMICOLON:toks) = comSeq toks
comSeq toks          = seqOpt (com toks)

seqOpt :: (Com, [Token]) -> (Com, [Token])

seqOpt (c, [])          = (c, [])
seqOpt (c, toks@(RBRACE:_) = (c, toks)
seqOpt (c, SEMICOLON:toks) = seqOpt (c :~: c', toks')

```

```

    where
      (c', toks') = comSeq toks

-- Parse a whole program -- no remaining tokens

-- program ::= comSeq

parseProg :: String -> Com

parseProg source = body
  where
    (body, []) = comSeq $ scan source

```

Interpreter.hs

```

module Interpreter where

import AbsSyn

-- A good match with the semantic definition would be to build the state
-- data structure as a function from Names to Ints:

{-
type State = Name -> Int

arid :: State
arid = \x -> error ("Unknown identifier: " ++ x)

update :: State -> Name -> Int -> State
update s x n = \y -> if x==y then n else s y
-}

-- But we want the interpreter to print out the state which is the result
-- of the interpretation so a "finite" data representation is more
-- appropriate.

-- State is a list of pairs (a graph of the function). The update
-- operation scans the list for the argument so it is always an O(n)
-- operation. It would probably be better (and a little easier) to just
-- stick it on the front of the list every time (i.e. a stack). Straight
-- list manipulation is easier, but I want a new type which can be an
-- instance of Show.

newtype State = State [(Name, Int)]

apply :: State -> Name -> Int

apply (State fm) x =
  case lookup x fm of
    Just n   -> n
    Nothing -> error ("Undeclared identifier: " ++ show x)

update :: State -> Name -> Int -> State

```

```

update (State fm) x n =
  State (listUpdate fm x n)
    where listUpdate [] x n = [(x,n)]
          listUpdate ((y,n'):ps) x n
            | x==y      = ((x,n):ps)
            | otherwise = ((y,n'): listUpdate ps x n)

instance Show State where
  showsPrec _ (State []) s = s
  showsPrec i (State ((x,n):ps)) s
    = concat [ show x, "\t-> ", show n, "\n", showsPrec i (State ps) s]

arid :: State
arid = State []

-- Arithmetic expressions:

eA :: Aexp -> State -> Int

eA (Num n) s      = n
eA (Var x) s      = apply s x
eA (a0 :+: a1) s  = n0 + n1
  where n0 = eA a0 s
        n1 = eA a1 s
eA (a0 :-: a1) s  = n0 - n1
  where n0 = eA a0 s
        n1 = eA a1 s
eA (a0 **: a1) s  = n0 * n1
  where n0 = eA a0 s
        n1 = eA a1 s

-- Boolean expressions:

eB :: Bexp -> State -> Bool

eB TrueLit s      = True
eB FalseLit s     = False
eB (a0 ==: a1) s   = n0 == n1
  where n0 = eA a0 s
        n1 = eA a1 s
eB (a0 <=: a1) s   = n0 <= n1
  where n0 = eA a0 s
        n1 = eA a1 s
eB (Not b) s      = not t
  where t = eB b s
eB (b0 'And' b1) s = t0 && t1
  where t0 = eB b0 s
        t1 = eB b1 s
eB (b0 'Or' b1) s  = t0 || t1
  where t0 = eB b0 s
        t1 = eB b1 s

-- Commands:

```

```

eC :: Com -> State -> State

eC (x := a) s      = update s x n
  where n = eA a s
eC Skip s          = s
eC (c0 :~: c1) s   = s'',
  where s' = eC c0 s
        s'' = eC c1 s'
eC (If b c0 c1) s
  | t           = s0
  | otherwise   = s1
  where t       = eB b s
        s0     = eC c0 s
        s1     = eC c1 s
eC (While b c) s
  | t           = s'',
  | otherwise   = s
  where t       = eB b s
        s'     = eC c s
        s''    = eC (While b c) s'

```

Main.hs

```

module Main (main) where

import IO
import Parser
import Interpreter

-- The usual general file opening function.

getAndOpenFile :: String -> IOMode -> IO Handle
getAndOpenFile prompt mode =
  do putStr prompt
     name <- getLine
     catch (openFile name mode)
           (\_ -> do putStrLn ("Cannot open " ++ name ++ "\n")
                    getAndOpenFile prompt mode)

-- Main function: get the source Imp file, parse it, interpret it and
-- print the resulting state to stdout.

main :: IO()
main = do
  sourceFile <- getAndOpenFile "Path of Imp source file: " ReadMode
  imp <- hGetContents sourceFile
  let result = eC (parseProg imp) arid
      putStr (show result)

```

Two Simple *Imp* Programs

swap.imp

```
{- A simple Imp program to swap the values in x and y
{- A few gratuitous syntactic and lexical wiggles

-} -}
```

```
x := 5; y := 3;
```

```
{z := x;
  x := y;
  y := z;; }
```

```
fact.imp
```

```
{- A simple Imp program to compute 10! -}
```

```
n := 10;
```

```
fact := 1;
while 0 <= (n-1) do {
  fact := fact * n;
  n := n-1
}
```