

Programming Language Implementation I

In this lecture and the next 9 lectures we will look at the implementation of programming languages, in particular at compilers.

- Main approaches to implementation, main phases in a compiler.
- Lexical analysis and regular expressions.
- Syntax analysis, context-free grammars and parsing
- Semantic analysis, attribute grammars, type inference and abstract interpretation.
- Machine independent optimisation
- Code generation, the runtime system, and peephole optimisation.

The material is (loosely) based on (Aho, Sethi and Ullman) and (Wilhelm and Maurer).

A useful source for compiler implementations (to obtain hands-on knowledge)
<http://www.idiom.com/free-compilers/>

Interpreters



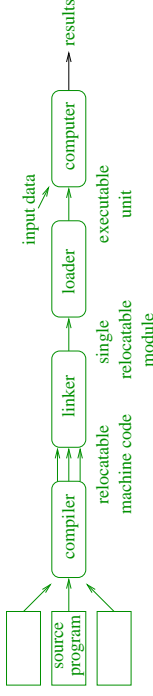
Programming language implementation varies between **interpretation** and **compilation**.

Programs can be run using an **interpreter**. This is a program which executes the program directly, acting as a software simulation of a computer which understands the high-level program constructs rather than machine instructions. I.e it implements a **virtual machine** for the language.

Basic and HTML are usually interpreted.

Interpretation supports interactive program development, provides better debugging support and run-time checks, but programs run significantly slower.

Compilers



A **compiler** implements a programming language on some target machine by **translating** it into a language that this machine “understands” directly.

Compilation into machine code was crucial to the acceptance of the first programming languages such as FORTRAN since it allowed high-level programs to run as fast as the equivalent assembly language program.

C and C++ are examples of languages which are usually compiled into machine code.

Programs run much faster if compiled rather than interpreted, but writing a good compiler is more difficult than writing an interpreter.

Compilers (Cont.)

However a compiler need not compile into machine code. It can translate the program into any other desired language. It is common to write compilers which compile into

- virtual/abstract machine code
- other programming languages as well as
- native machine code

This means compilation may be multi-phase. Eg. Early compilers for C++ compiled into C, then used the C compiler to generate the machine code.

Abstract Machines/Virtual Machines



Nowadays the target of compilation is often a **virtual machine** or **abstract machine**. A virtual machine is essentially an interpreter for a virtual language that runs on the target machine.

For instance Prolog is often compiled into the WAM (Warren Abstract Machine) code, and Java into JVM (Java Virtual Machine) code.

Abstract Machines/Virtual Machines (Cont.)

The virtual language provides an intermediate level between high-level language and native machine code that is specifically designed for a particular class of languages. (e.g. procedural, object-oriented, logic, functional) and provides the basic data structures and basic control mechanisms in these languages.

Since the virtual machine language is closer to the high-level language, translation into virtual machine code is easier. Compilation into native machine code is more complex, since machine code requires explicit data structure and address management, provides only very simple forms of control structures and requires more optimization.

The main arguments for using a virtual machine (such as the JVM) are

- ease of implementation
- increased portability
- security and better run-time checking (easier encapsulation)

However, native code is typically much faster.

Compilation

Compilation is one of the best understood areas of computer science:

- Useful general principles of compilation,
- A well understood underlying theory and
- several useful tools such as Flex and bison which are based on this theory and help automate compiler construction.

In the remainder of this subject we shall examine compilation.

Compilation (Cont.)

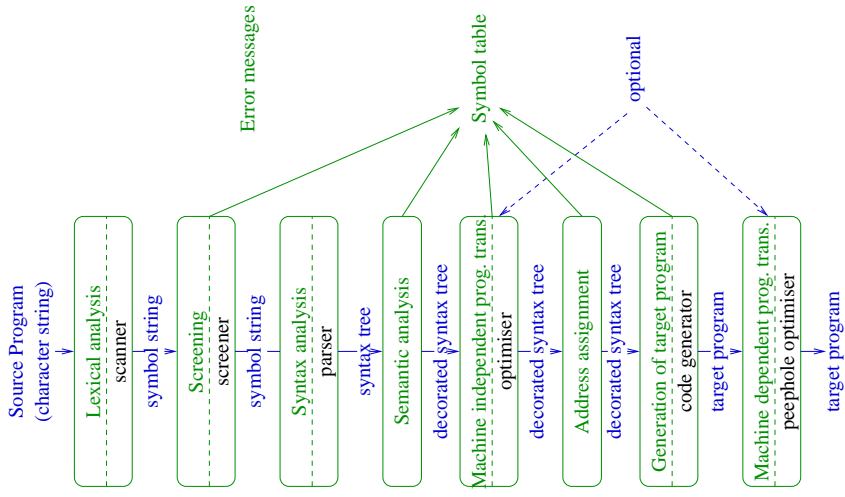
Many techniques used in compiler construction are general techniques for translation and are useful for implementing more than just programming languages, for example

- document formatting languages, such as PostScript, HTML or \LaTeX ,
 - VLSI layout languages,
 - interpreting command line arguments,
 - XML readers
- ...

Apart from compilers they are useful for building programming tools like

- debuggers,
- syntax-based editors,
- interpreters,
- macro preprocessors and other preprocessors,
- X referencers,
- translators such as HTML2TeX etc
- silicon compilers ...

Phases of a Compiler



Symbol Table and Parse Tree

The two main data structures in a compiler are

- the **parse tree**, which reflects the **hierarchical program structure**.
- the **symbol table** which contains
 - identifiers from the screener
 - types and other information for identifiers from semantic analysis
 - addresses for variables and functions from address allocation

Lexical Analysis

The scanner carries out the lexical analysis of the program, transforming the input character string into a sequence of lexical units ie tokens.

Typical tokens are:

- identifiers (together with the string),
- punctuation marks such as “;”,
- 1-2 character operators, such as “=”, “*”, “(”, “)”
- integer, real, character, boolean, string etc constants.

It often removes whitespace (newlines, tabs and spaces) and may remove comments.

Lexical Analysis

Example

```
strcpy(s, t)
char *s, *t;
{ while(*s++ = *t++); }
```

```
[ strcpy, (, s, ,, t, ), char, *, s, ,, *, t, ;, {,
while, (, *, s, +, +, =, *, t, +, +, ), ;, } ]
```

Methods

For the specification of the token-level we use regular expressions.

Recognition of these can be implemented with finite state automata (FSA).

Screening

The scanner performs further analysis and classification of the tokens. It typically does the following

- Recognizes **reserved words** such as `var` or `fun` among the identifiers.
- Removes symbols that are **irrelevant** for subsequent processing, ie removes comments.
- Recognizes symbols that are not part of the program but are directives to the compiler (called **pragmas**).
- Puts identifiers into the symbol table and replaces the identifier name with, say, an index into the table.

Usually the scanner and screener are combined into a single module (usually called the scanner) but conceptually two different phases are being performed.

The scanner may operate as a **separate pass** or it may act on **demand** from the parser.

Screening

Example

```
[ strcpy, (, s, ,, t, ), char,
*, s, ,, *, t, ,, {,
while, (, *, s, t, +, =,
*, t, +, +, ), ;, } ]
```

```
[ res(strlen), (, id(1), ,, id(2), ),
res(char), op(*), id(1), ,, op(*), id(2), ;,
{, res(while), (, *, ... ]
```

Syntax Analysis

The parser performs syntax analysis. It knows the structure of expressions, statements, declarations etc in the programming language and uses this to determine the hierarchical syntactic structure of the program.

It generates a syntax tree which is (together with the symbol table) the central data structure for all following phases.

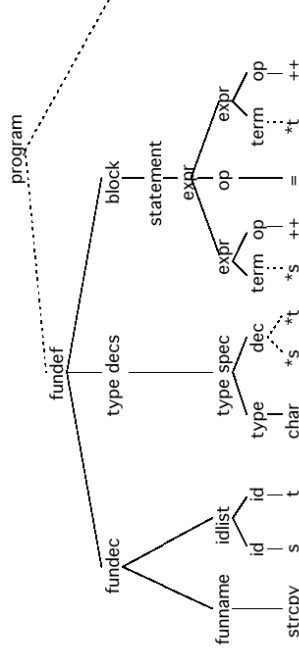
An important role of the parser is to detect, locate, diagnose and even fix errors in the syntax of the program. Correct detection of (and recovery from) syntax errors is one of the more difficult tasks in syntax analysis.

The area of syntax analysis and parsing is one of the best studied fields in computer science. The theory is based on context-free grammars and pushdown automata, but it makes use of several refined classes of grammars.

Example Parse Tree

The parse tree (syntax tree) reflects the hierarchical structure of the program. Example

```
[ res(strlen), (, id(1), ,, id(2), ),  
res(char), op(*), id(1), ,, op(*), id(2), ;;  
{, res(while), (, *, ...]
```



Semantic Analysis

Determines those non-syntactic properties that can be determined from the program text.

Typically determines the kind of each identifier. For instance, which identifiers are variables, and which are functions.

Performs type checking and type inference.

Checks that variables and procedures etc are defined before use.

Adds the kind and type information to the symbol table.

Machine-Independent Optimization

This uses static analysis to do more checking and to perform efficiency increasing transformations on the level of the source program code.

One example of checking is to determine that on every possible execution path each variable is initialized before being used.

Typical code optimizations include:

- Calculation of sub-expressions whose values are already known.
- Extraction of loop-invariant computation from loops.
- Elimination of redundant computations.
- Elimination of dead code.
- Procedure call inlining.
- Tail-recursion optimization.

This phase is optional.

Address Assignment

This performs storage allocation and address assignment.

This requires information about the target machine such as the

- word length
- address length
- types of access instructions
- alignment conditions.

Variable addresses are placed in the symbol table.

Generation of the Target Program

The code generator generates the target code. Issues are

- **instruction selection**
Eg. how is $a := a+1$ implemented (via addition ADD or via increment INC?)
- **register allocation**
Which variables will be assigned to processor registers and at which point of the program to which register will each of these be assigned?
This is complicated because not all registers can perform all functions, e.g. special registers used for multiplication, results are often returned in fixed register
- **instruction scheduling**
If the target machine has parallel processing capabilities then the instruction sequence must be scheduled so that execution is correct and that parallel-processing is maximised.

Example

Suppose that the target machine has instructions:

LOAD *A, R* Load contents of location with address *A* into register *R*

STORE *A, R* Store contents of register *R* into location with address *A*

LOADI *I, R* Load integer constant *I* into register *R*

MULT *A, R* Multiply the contents of location with address *A* by the contents of register *R* and store the result in *R*

And that the machine has two registers, *R1* and *R2*.

Exercise: Give target code for the example program.

Machine-Dependent Code Improvement

In contrast to machine-independent optimization which looks at more global properties, machine-dependent optimization tries to exploit properties of the target architecture to make the generated code more efficient.

This is usually done by using **peephole optimization**.

Peephole optimisation passes a small “window” over the target code and tries to replace the visible instruction sequence by a better one.

This typically

- eliminates **useless instructions**,
- replaces **general instructions** by more efficient **specialized instructions**.

Of course this phase is **optional**.

Summary

We have looked at

- The function of compilers and interpreters: translation vs. execution
- Virtual machines
- The basic structure and processing phases of a compiler
 - Lexical analysis
 - Screening
 - Syntax analysis
 - Semantic analysis
 - Machine independent optimisation
 - Address assignment
 - Generation of target program
 - Machine dependent optimisation
- The main data structures used in a compiler: symbol tables and parse trees

In the next few weeks we will look at each stage in more detail

Homework

- Read Chapter 6 of Wilhelm and Maurer.
- Revise regular expressions. In particular you should revise
 - regular expressions
 - finite state automata (FSA)
 - conversion of a non-deterministic FSA into a deterministic FSA.

Programming Language Implementation II

In this lecture we will look at lexical analysis, scanning and screening.

- Regular languages and regular expressions.
- Finite state automata (FSA).
- Deterministic FSAs.
- Flex.

The material is (loosely) based on Wilhelm and Maurer Chapter 7.

Scanners and Screeners

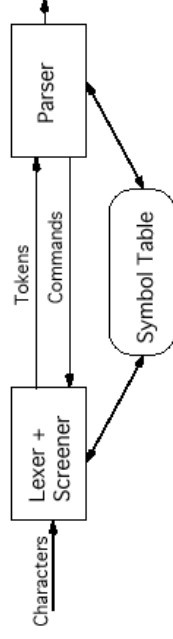
Recall that the **scanner** carries out the lexical analysis of the program, transforming the input character string into a sequence of lexical units ie **tokens**. Typical tokens are:

- identifiers (together with the string),
- punctuation marks such as “;”,
- 1-2 character operators, such as “=”, “*”, “(”, “)”
- integer, real, character, boolean, string etc constants.

The **screener** performs further analysis and classification of the tokens. It typically

- Recognizes reserved words
- Removes symbols that are irrelevant for subsequent processing,
- Recognizes pragmas.
- Puts identifiers into the symbol table and links the tokens to the entries in the symbol table.

Scanners and Screeners (Cont.)



Typically the scanner and screener are combined into one module called the scanner. This may operate as a **separate pass** or, more commonly, it may act **on demand** from the parser:

Lexical analysis is based on the **theory of regular expressions**.

Words and Languages

A **word** x over an **alphabet** Σ is a finite sequence of characters from (the set of characters) Σ .

We let ϵ be the **empty word**, ie the null string.

A **(formal) language** L over Σ is a set of words over Σ .

We will use the **following operations on formal languages**:

- $L_1 \cup L_2$ **Union of languages**
- $L_1 L_2$ **Concatenation of languages**
- L^n $\{x_1 \dots x_n \mid x_i \in L, 1 \leq i \leq n\}$
- L^* $\cup_{n \geq 0} L^n$ **Closure of a language**
- L^+ $\cup_{n > 0} L^n$
- \bar{L} $\Sigma^* - L$ **Complement of a language**

Let $D = \{0, 1, \dots, 9\}$ **and** $L = \{a, \dots, z, A, \dots, Z\}$. **Then:**

- $L \cup D$ **is the set of letters and digits**
- $LD = \{a0, \dots, a9, \dots, Z0, \dots, Z9\}$
- D^+ **are the strings of digits.**

Exercise: What are $L(L \cup D)^*$ and $(D - \{0\})D^*$?

Regular Languages and Regular Expressions

The regular expressions over alphabet Σ are:

- ϵ , which describes the language $\{\epsilon\}$
- a for any $a \in \Sigma$, which describes the language $\{a\}$
- If regular expressions r and s describe the languages R and S then,
 - $(r|s)$ is a regular expression describing $R \cup S$
 - (rs) is a regular expression describing RS
 - (r^*) is a regular expression describing R^* .

A language which can be exactly described by a regular expression is called **regular**.

Many useful languages are not regular, eg the set of palindromes.

* has highest precedence, followed by concatenation, then disjunction.

Finite-State Automata

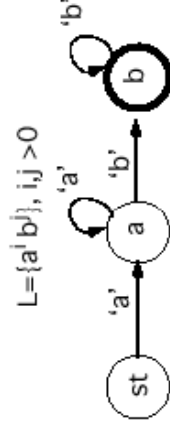
A finite state automata can be used to recognize if a string is in a regular language or not.

A (non-deterministic) finite state automata (NFA) consists of an:

- input alphabet Σ ,
- a finite set of states Q ,
- an initial state $q_0 \in Q$,
- a set of final states $F \subseteq Q$, and
- a transition relation $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$.

A NFA accepts (or recognizes) words for which it has a path from the initial state to a final state.

We usually represent an NFA using a state transition diagram. E.g.



Finite-State Automata (Cont.)

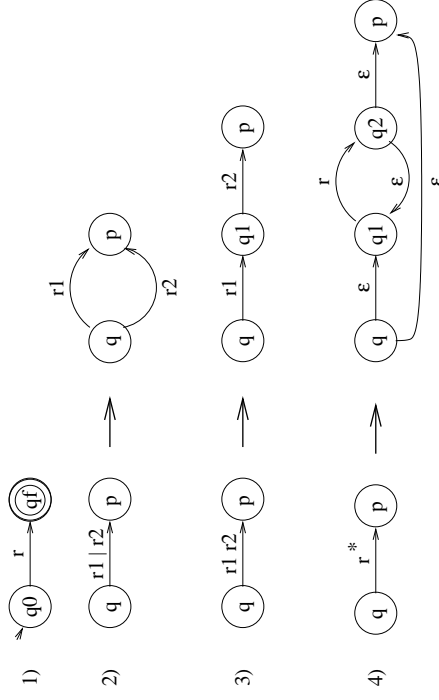
- An example NFA has
- input alphabet $\{0, 1, \dots, 9, \epsilon, E\}$,
 - states $\{0, \dots, 7\}$,
 - initial state 0,
 - final states $\{1, 7\}$, and
 - transition table,

	$0, 1, \dots, 9$	ϵ	E	ϵ
0	\emptyset	\emptyset	\emptyset	\emptyset
1	$\{1\}$	\emptyset	\emptyset	\emptyset
2	$\{2\}$	$\{3\}$	\emptyset	\emptyset
3	$\{4\}$	\emptyset	\emptyset	\emptyset
4	$\{4\}$	\emptyset	$\{5\}$	$\{7\}$
5	$\{6\}$	\emptyset	\emptyset	\emptyset
6	$\{7\}$	\emptyset	\emptyset	\emptyset
7	\emptyset	\emptyset	\emptyset	\emptyset

What does this recognise?

Finite-State Automata (Cont)

It is straightforward to give a NFA for recognizing words in the language of an arbitrary regular expression. The rules are



Example

Exercise: What is a NFA for recognizing the language of $a^*(a|b)^*$?

Deterministic Finite-State Automata

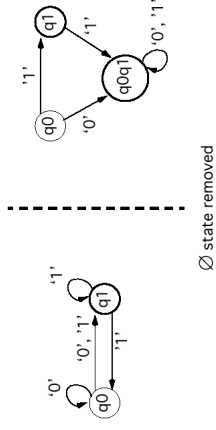
A finite state automata is deterministic if it has no transitions via ϵ and at most one successor state for each pair (q, a) where $q \in Q$ and $a \in \Sigma$. We call such an automata a DFA.

Every NFA has a corresponding DFA which recognizes the same language(!)

Converting a NFA to a DFA

Example (Cont.)

This can be generated automatically using the subset construction algorithm:



What is a DFA for recognizing the language of $a(a|b)^*$?

The DFA has states corresponding to sets of states in the original NFA.

We start from the new “super” start state which is the old start state + all states which are ϵ successors.

The successor state of a super state S for character a is the super state containing all states which can be reached from a state in S under a and adding their ϵ successor states.

Any super state which contains a final state is a final state.

The only problem is that this may lead to an exponential increase in the number of states.

Minimizing DFAs

The DFA for recognizing the regular language generated by the above two steps is usually not the smallest DFA recognizing the language.

The DFA minimization algorithm takes a DFA and returns a DFA with the smallest number of states which recognizes the same language.

The key idea is to partition the original states into states which have equivalent accepting behaviour.

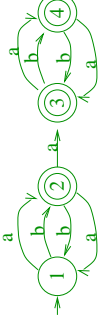
Partition the states into the final states and the non-final states.

Repeatedly split a partition if its states have different accepting behavior for some character in the alphabet, until all states in each partition have the same behaviour.

Finally, remove dead (no path to final state) and unreachable (no path from start state) states.

Minimizing DFA Example

Find the minimal DFA for



Exercise: What is the minimal DFA for $a(a|b)^*$?

Scanner Generators

A scanner generator automates the process of writing an efficient scanner from a regular expression description of the language tokens.

Basically a scanner generator does the following:

Input: Regular expressions R_1, \dots, R_n describing the tokens of interest T_1, \dots, T_n .
Combine R_1, \dots, R_n into a single regular expression R .
Compute the corresponding NFA for R .
Transform this into a DFA.
Minimize the DFA.
Output: An efficient scanning function based on the minimal DFA.

When the generated scanner is used, it begins with the first character that has not yet been **consumed**.

The generated DFA is then used to process the characters.

The scanner reports that it has found a symbol when it is in a final state and there is no transition using the next character. If there is no transition from the actual state and the actual state is not a final state, it **backtracks** to the last final state it went through. If there is no such state, an error has occurred.

Flex

flex is a public domain scanner generator for UNIX systems. **flex** provides an interface to C so that the programmer can directly program the desired screening functionality into the scanner itself.

It also has a (large) number of extensions to standard regular expressions to make the task of specification easier.

flex takes a lexical specification and produces a C file `lex.yy.c`. This contains the function `yylex` which can be called by other programs such as a parser. The last symbol found can be accessed via the global character pointer `yytext` and its length is in `yylen`.

Flex (Cont.)

A flex specification has the form

```
<Definitions>
%%
<Rules>
%%
<The user's C functions>
```

The **definitions** section introduces identifiers for regular expressions. These behave just like macro definitions.

The **rules** consist of a **pattern** (basically a regular expression) and an **action** which is the piece of C code to be performed if the pattern is matched.

The **user code** is copied verbatim to `lex.yy.c`, as is indented text in the rules and declarations section and text between `%{` and `%}` where these occur unindented on a line by themselves.

Simple Flex Example

The following flex specification `count.flex` counts the number of characters and lines.

```
/* count number of lines and
   characters in stdin */
int num_lines = 0, num_chars = 0;

%%
\n ++num_lines; ++num_chars;
. ++num_chars;

%%
main()
{
    yylex();
    printf("# of lines= %d, # of characters= %d\n",
           num_lines, num_chars);
}
```

We generate the scanner as follows:

```
% flex count.flex
% gcc lex.yy.c -lfl
cfe: Error: count.flex, line 6: 'num_lines' undefined;
reoccurrences will not be reported.
++num_lines; ++num_chars;
--
cfe: Error: count.flex, line 6: 'num_chars' undefined;
reoccurrences will not be reported.
++num_lines; ++num_chars;
-----
cfe: Error: count.flex, line 14: 'num_lines' undefined;
reoccurrences will not be reported.
num_lines, num_chars);
-----
cfe: Error: count.flex, line 14: 'num_chars' undefined;
reoccurrences will not be reported.
num_lines, num_chars);
-----
```

Note that the error line numbers refer to the original flex specification.

After fixing the error, we now have:

```
% flex count.flex
% gcc lex.yy.c -lfl
% a.out < count.flex
# of lines= 15, # of characters= 250
%
```

Patterns in Flex

The patterns are an extended set of regular expressions:

<code>x</code>	match the character 'x'
<code>.</code>	any character except newline
<code>[a-zA-Z]</code>	a character class, in this case any letter
<code>[^A-Z]</code>	a negated character class
<code>r*</code>	zero or more r's
<code>r+</code>	1 or more r's
<code>r?</code>	0 or 1 r
<code>{NAME}</code>	the expansion of the NAME definition
<code>"[xyz]"foo"</code>	a literal string, in this case "[xyz]"foo"
<code>\X</code>	the ANSI-C interpretation of \X
<code><<EOF>></code>	EOF
<code>(r)</code>	the regular expression r
<code>rs</code>	concatenation of r and s
<code>r s</code>	an r or an s
<code>r/s</code>	an r followed by the trailing context s
<code>^r</code>	an r at the start of the line
<code><s>r</code>	an r but only when the condition s is activated

See the online documentation for more details.

When pattern matching the generated scanner chooses the pattern which matches the longest string of characters, and if more than one pattern matches the same length string it chooses the first pattern in the specification.

A Bigger Example

```
/* scanner for Pascal */
/* definitions */
%{
#include <math.h>
%}
DIGIT [0-9]
ID [a-z][a-z0-9]*
WSPACE [ \t\n]
OP ("+"|"-"|"*"|" "/" | "=")
PUNC (","|" ";" | ":" | "(" | ")" )
%%
/* rules */
{DIGIT}+ {
printf("An integer: %s(%d)\n",yytext,atoi(yytext));}
{DIGIT}+ {
printf("A float: %s(%g)\n",yytext,atof(yytext));}
{OP} { printf("An operator: %s\n",yytext); }
{PUNC} { printf("Punctuation: %s\n",yytext);}
if[then|begin|end|procedure|function|var {
printf("A reserved word: %s\n",yytext);}
{ID} { printf("An identifier: %s\n",yytext);}
"{"("[^}]*"}" /* discard comments */
{WSPACE}+ { printf("Separator\n"); }
. printf("Error unexpected character: %s\n",yytext);
}
/* Auxiliary C code */
main()
{ yylex(); }
```

45

If **toy.pl** is the file containing

```
function(int x): int;
begin
  {increment x}
  x := x+1;
end
```

then we have

46

Summary

```
% flex cascal.flex
% gcc lex.yy.c -lfl
% a.out < toy.pl
A reserved word: function
Punctuation: (
An identifier: int
Separator
An identifier: x
Punctuation: )
Punctuation: ;
Separator
An identifier: int
Punctuation: ;
Separator
A reserved word: begin
Separator
Separator
An identifier: x
Separator
An operator: :=
Separator
An identifier: x
An operator: +
An integer: 1(1)
Punctuation: ;
Separator
A reserved word: end
Separator
```

We have looked at lexical analysis:

- Regular languages and expressions.
- Nondeterministic and deterministic finite state automata.
- Computing a minimal DFA to recognize a given regular language.
- Scanner generators.
- flex.

Homework

- Read Chapter 7 of Wilhelm and Maurer.
- Give a corresponding NFA to $(a|b)^*abb$. Convert this to a DFA. Now minimize the DFA.
- Get the online flex documentation.
- Use flex to generate a program which counts the number of lines, characters and words in standard input.

Programming Language Implementation III

In this lecture we will look at syntax analysis. i.e. parsing.

- Context free grammars.
- Recursive descent parsing.
- Removing left recursion and left factoring.

The material is (loosely) based on Aho et al Chapter 4.

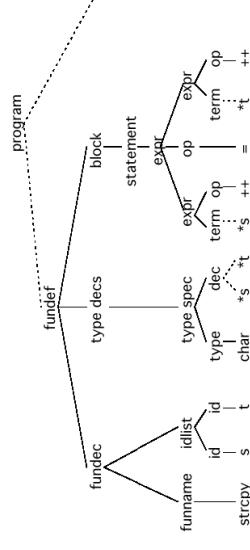
Review: Syntax Analysis

The task of syntax analysis is to determine the hierarchical syntactic structure of the program from the stream of tokens returned by the screener.

```
[ strcpy, (, s, ,, t, ), char,  
*, s, ,, *, t, ,, {,  
while, (, *, s, +, +, =,  
*, t, +, +, ),;, } ]
```

```
[ res(strlen), (, id(1), ,, id(2), ),  
res(char), op(*), id(1), ,, op(*), id(2), ;,  
{, res(while), (, *, ... ]
```

The parse tree (syntax tree) reflects the hierarchical structure of the program.



Why Not Use Regular Expressions?

Most programming language constructs have an inherently recursive structure.

For example, if S_1 and S_2 are statements and E is an expression then

if E then S_1 else S_2

is a statement.

This form of recursive statement cannot be specified using a regular expression—something more powerful is needed.

Review of Context Free Grammars

A (context-free) grammar G consists of:

- A set of terminal symbols (also called tokens), T ;
- A set of non-terminal symbols, N ;
- A start symbol, $S \in N$;
- A set of production (rules) each of form $X \rightarrow \alpha$ where $X \in N$ and α is a sequence from $(N \cup T)^*$.

The following grammar defines a subset of arithmetic expressions. It has terminal symbols,

`int + * ()`

and non-terminal symbols exp , $term$ and $factor$ where exp is the start symbol and the productions are:

```
exp  → term
exp  → exp + term
term → factor
term → term * factor
factor → int
factor → (exp)
```

Usually we combine productions with the same LHS symbol A into alternatives for A , thus we write

$exp \rightarrow term \mid exp + term$

Language and Parse Trees

We let α, β, γ stand for sentences, that is sequences from $(N \cup T)^*$.

$\alpha \Rightarrow \beta$ indicates that β can be derived from α by using a single production rule.

$\alpha \Rightarrow^* \beta$ indicates that β can be derived from α by using zero or more applications of a production rule.

The language generated by grammar G with start symbol S is $\{\alpha \mid \alpha \in T^* \text{ and } S \Rightarrow^* \alpha\}$.

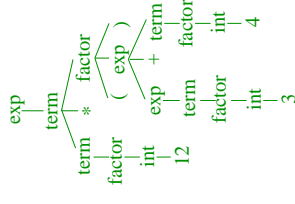
For instance, $12 * (3 + 4)$ (more exactly $int * (int + int)$) is in the language of our example.

Parse Trees

A derivation can be represented using a parse or syntax tree.

- each leaf is a terminal
- the leaves (in pre-order) are the input sequence
- each inner node is a non-terminal
- the root is the start symbol of the grammar

Consider the derivation for $12 * 3 + 4$:



Notice that the parse tree abstracts away from the order in which production rules are used.

Structure Trees

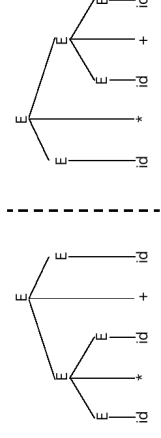
In practice parsers usually build a **structure tree** which only contains the essential structure of the parse tree.

Ambiguity

A grammar is **ambiguous** if some sentences have more than one parse tree.

For example, the following grammar is ambiguous:

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$



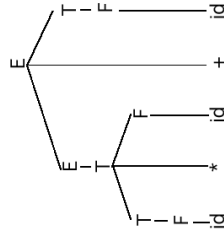
Indeed, in some simple applications even a structure tree need not be built. However, conceptually the role of the parser is to build the parse tree.

Ambiguity (Cont.)

In this case, we can modify the grammar so that it is no longer ambiguous

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

This grammar allows only a single parse tree.



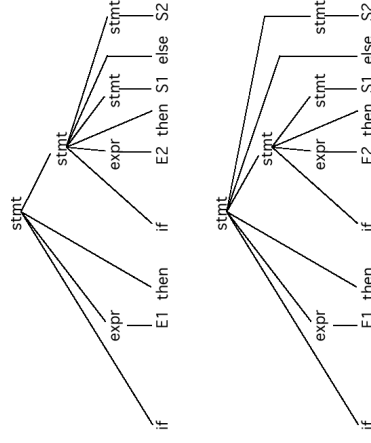
Dis-Ambiguation

We cannot always remove ambiguity: Some context-free languages are **inherently ambiguous**.

However as we have seen in many other cases it can be eliminated. As another example consider

$$\begin{aligned} stmt &\rightarrow \text{if } exp \text{ then } stmt \text{ else } stmt \\ &\mid \text{if } exp \text{ then } stmt \\ &\mid \text{other} \end{aligned}$$

This is ambiguous since
if e1 then if e2 then s1 else s2
has two parse trees.



Dis-Ambiguation (cont.)

Disambiguation requires knowledge about the intended structure of the language.

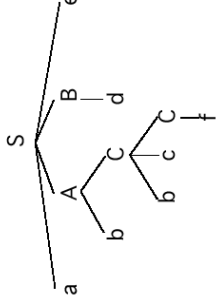
If we choose to associate the dangling else with the closest previous unmatched then, we can use the grammar

```
stmt      → matched
           | unmatched
matched   → if exp then matched else matched
           | other
unmatched → if exp then stmt
           | if exp then matched else unmatched
```

Top-Down Parsing

In top-down parsing the parse tree is built from the root downwards.

```
S → aABe
A → bC
B → d
C → bcC | f
```



A **leftmost (rightmost)** derivation always replace the **leftmost (rightmost)** non-terminal in the current sentence.

For our example the leftmost derivation is:

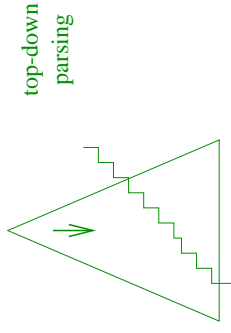
$S \Rightarrow aABe \Rightarrow abCBe \Rightarrow abbcCBe \Rightarrow abbcfBe \Rightarrow abbcfde.$

And the rightmost is:

$S \Rightarrow aABe \Rightarrow aAde \Rightarrow abCde \Rightarrow abbcCde \Rightarrow abbcfde.$

Predictive Top-down Parsing

In most top-down parsing algorithms the parse tree is built from the root downwards and the parser scans the input-left-to-right one token at a time, essentially constructing a leftmost derivation



If the parser is always able to guess which rule to use then it is **deterministic** otherwise it is **non-deterministic** and will need to employ backtracking.

If it is able to guess which rule to use it is said to be a **predictive** parser.

We will look at **recursive descent** and **table-driven predictive** parsers.

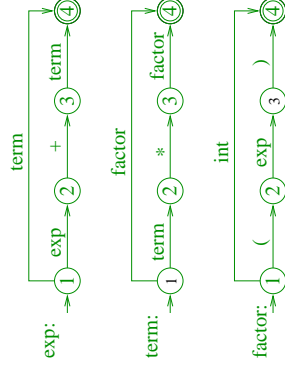
Transition Diagrams

It is useful to represent the grammar using a **transition diagram** for each non-terminal symbol.

Recall the grammar

$$\begin{aligned} \text{exp} &\rightarrow \text{term} \mid \text{exp} + \text{term} \\ \text{term} &\rightarrow \text{factor} \mid \text{term} * \text{factor} \\ \text{factor} &\rightarrow \text{int} \mid (\text{exp}) \end{aligned}$$

The transition diagram for it is:



Recursive-Descent Parsing

The idea behind recursive-descent parsing is to write mutually recursive functions, one for each non-terminal symbol, which mirror the grammar.

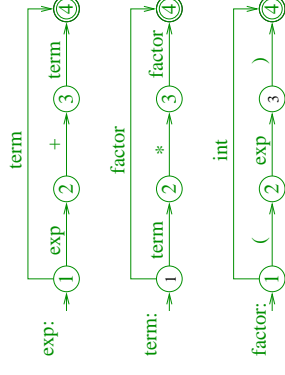
A recursive-descent parser works off the transition diagram as follows.

It begins from the start state for the start symbol.

Now if it is in state s with an edge labelled by symbol x going to state t it does the following:

- If x is the terminal symbol a and the next input symbol is a , then the parser moves the input cursor one symbol right (ie consumes a) and goes to state t .
- If x is the non-terminal symbol A , the parser calls itself recursively, beginning from the starting state of A . If it ever reaches the final state for A it returns and immediately moves to state t , in effect having read A from the input stream.
- Or if x is ϵ it simply moves to state t without reading any input. This is only done if no other transition from s can be performed.

Recursive-Descent Parsing: Example



Does the previous transition diagram lend itself to recursive-descent parsing?

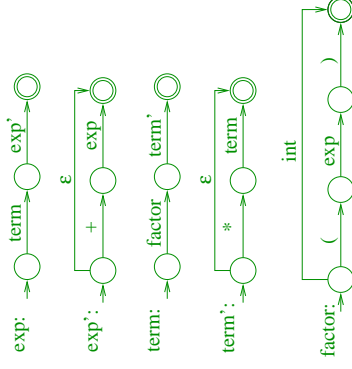
Recursive-Descent Parsing Example

Recursive-Descent Parsing Example (Cont.)

A better grammar for recursive descent parsing of arithmetic expressions is

$$\begin{aligned} \text{exp} &\rightarrow \text{term } \text{exp}' \\ \text{exp}' &\rightarrow +\text{exp} \mid \epsilon \\ \text{term} &\rightarrow \text{factor } \text{term}' \\ \text{term}' &\rightarrow *\text{term} \mid \epsilon \\ \text{factor} &\rightarrow \text{int} \mid (\text{exp}) \end{aligned}$$

The transition diagram for it is:



This can be used as the basis for a recursive descent parser.

The flex program for the lexical analyser is

```

/* definitions */
enum symbols {INT, PLUS, MULT, LPAR, RPAR, END} lookahead;

DIGIT [0-9]
WSPACE [ \t\n]
%%
/* rules */
{DIGIT}+ lookahead = INT; return;
"+" lookahead = PLUS; return;
"*" lookahead = MULT; return;
"(" lookahead = LPAR; return;
")" lookahead = RPAR; return;
<<EOF>> lookahead = END; return;
{WSPACE}+ /*discard whitespace*/
%%
/* definitions */
void consume(void) {
    printf("Consuming %s \n", yytext);
    yylex();
}
  
```

The parser proper is:

```

int exp(void) {
    return term() && exp1();
}
int exp1(void) {
    if(lookahead == PLUS) {
        consume();
        return exp(); }
    else return 1;
}
int term(void) {
    return factor() && term1();
}
int term1(void) {
    if(lookahead == MULT) {
        consume();
        return term(); }
    else return 1;
}
int factor(void) {
    int i;
    switch (lookahead) {
    case LPAR :
        consume();
        i = exp();
        if(lookahead == RPAR) {
            consume();
            return i; }
        else return 0;
    case INT :
        consume();
        return 1;
    default :
        return 0;
    }
}
}

```

```

main() {
    yylex();
    if (exp() && lookahead == END)
        printf("successful parse\n");
    else
        printf("syntax error\n");
}

```

Some examples:

```

% a.out
( 345 + 54 ) * (9+0) + 15
^D
Consuming (
Consuming 345
Consuming +
Consuming 54
Consuming )
Consuming *
Consuming (
Consuming 9
Consuming +
Consuming 0
Consuming )
Consuming +
Consuming 15
successful parse

% a.out
345 * +2
Consuming 345
Consuming *
syntax error

```

Elimination of Left Recursion

The first example transition diagram was not suitable for recursive-descent parsing because it is left-recursive that is, for some non-terminal A ,

$$A \Rightarrow \dots \Rightarrow A\alpha$$

for some string α .

Top-down parsing methods cannot handle grammars with left-recursion. Fortunately, we can always remove left-recursion.

In the simple case we have **direct recursion**, say with the rule

$$A \rightarrow A\alpha \mid \beta.$$

This can be replaced by the non-left-recursive productions

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Elimination of Left Recursion (Cont.)

Using this transformation we can eliminate left-recursion from the grammar:

$$\begin{aligned} \text{exp} &\rightarrow \text{term} \mid \text{exp} + \text{term} \\ \text{term} &\rightarrow \text{factor} \mid \text{term} * \text{factor} \\ \text{factor} &\rightarrow \text{int} \mid (\text{exp}) \end{aligned}$$

We get the grammar:

Elimination of Left Recursion (Cont.)

More generally, direct recursion can be removed from the rule

$$A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

by replacing it by the productions

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \dots \mid \beta_m A' \\ A' &\rightarrow \alpha_1 A' \mid \dots \mid \alpha_n A' \mid \epsilon \end{aligned}$$

We can also remove indirect recursion using the following algorithm.

```
Arrange the non-terminals in some order  $A_1, \dots, A_n$ .
for  $i = 1, \dots, n$  do
  for  $j = 1, \dots, i - 1$  do
    replace each production of form  $A_i \rightarrow A_j \gamma$ 
    by the production
       $A_i \rightarrow \delta_1 \gamma \mid \dots \mid \delta_k \gamma$ 
    where  $A_j$  currently has the production
       $A_j \rightarrow \delta_1 \mid \dots \mid \delta_k$ 
  endfor
  eliminate direct left recursion from the  $A_i$  productions
endfor
```

Elimination of Cycles and ϵ -Productions

The above algorithm assumes that the input grammar has no cycles or ϵ -productions. So do practically all parsing algorithms. Luckily, these can be eliminated automatically.

A context-free grammar can be made ϵ -free (i.e. it does not contain any ϵ -productions) if the language generated by G does not contain ϵ . We will see in a later lecture how to do this.

A grammar has a cycle if $A \rightarrow \dots \rightarrow A$ for some non-terminal A . Clearly, a cycle in a derivation performs no useful function. How can cycles be removed automatically?

Left Factoring

Left factoring is another grammar transformation which is useful to produce a grammar suitable for predictive parsing.

The key idea is that when it is not clear which of two alternative productions to use to expand a non-terminal A , rewrite the productions so as to delay the commitment to one or the other.

For example, consider the grammar

$$\begin{array}{l} stmt \rightarrow \text{if } exp \text{ then } stmt \text{ else } stmt \\ \quad | \text{if } exp \text{ then } stmt \\ \quad | \text{other} \end{array}$$

if we encounter an if which production do we use?

Left Factoring (Cont.)

In general if a nonterminal A has productions some of which share a non-trivial common prefix $\alpha \neq \epsilon$ then

$$A \rightarrow \alpha\beta_1 \mid \dots \mid \alpha\beta_n \mid \gamma_1 \mid \dots \mid \gamma_m$$

can be rewritten to

$$\begin{array}{l} A \rightarrow \alpha A' \mid \gamma_1 \mid \dots \mid \gamma_m \\ A' \rightarrow \beta_1 \mid \dots \mid \beta_n \end{array}$$

Repeated application of this rewriting will ensure that no alternatives share a common prefix.

The above grammar will be rewritten to:

$$\begin{array}{l} stmt \rightarrow \text{if } exp \text{ then } stmt \text{ else } stmt \\ \quad | \text{other} \\ elsestmt \rightarrow \text{else } stmt \\ \quad | \epsilon \end{array}$$

Summary

We have looked at syntax analysis:

- Context free grammars.
- Recursive descent parsing.
- Removing left recursion and left factoring.

Homework

- Read Sections 4.2, 4.3, and 4.4 of Aho et al.
- Modify the grammar for arithmetic expressions to allow functions such as *sin* and *cos*.
- Build a recursive descent parser for this extended grammar.
- Modify your parser so that it computes the value of the expression, as long as it is syntactically valid.
- Use the left-recursion algorithm to remove recursion from the grammar

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sa \end{aligned}$$

Programming Language Implementation IV

In this lecture we will look at syntax analysis. i.e. parsing.

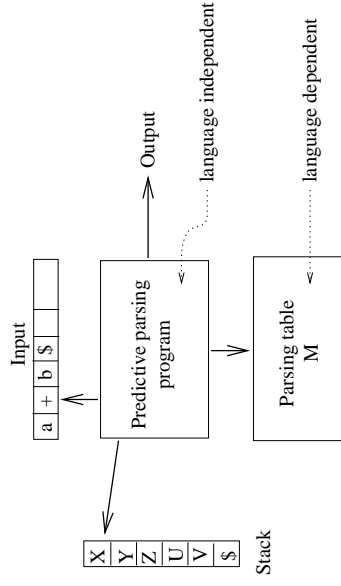
- Table driven predictive parsing.
- LL(1) grammars.

The material is (loosely) based on Aho et al Chapter 4.

Table-driven Predictive Parsing

It is possible to build a non-recursive predictive parser by maintaining a **stack** explicitly rather than implicitly via recursive calls.

In particular we can mechanically derive a **table driven** predictive parser from the grammar. This has the form



The **parsing table** is a 2-D array of entries $M[X, a]$ where X is a non-terminal and a is a terminal symbol or the special symbol $\$$ indicating end-of-input.

The parser is controlled by a simple program which does the following:

Initialize the stack to contain the start symbol on top of $\$$.

At each point the program considers the symbol on top of the stack X and a the current input symbol. There are 3 possibilities:

- If $X = a = \$$, the parser halts accepting the string.
- If $X = a \neq \$$, the parser pops X off the stack and “consumes” a , advancing to the next input symbol.
- If X is a non-terminal, the program looks up the parsing table entry $M[X, a]$. If this entry is a production of the form

$$X \rightarrow X_1 \dots X_n$$

then pop X from the stack and push X_n, X_{n-1}, \dots, X_1 on to the stack and output

$$X \rightarrow X_1 \dots X_n.$$

Otherwise if $M[X, a]$ is empty report an error.

Example

Recall the grammar

- $exp \rightarrow term\ exp' \quad (P1)$
- $exp' \rightarrow +exp \quad (P2)$
- $exp' \rightarrow \epsilon \quad (P3)$
- $term \rightarrow factor\ term' \quad (P4)$
- $term' \rightarrow *term \quad (P5)$
- $term' \rightarrow \epsilon \quad (P6)$
- $factor \rightarrow \mathbf{int} \quad (P7)$
- $factor \rightarrow (exp) \quad (P8)$

The parsing table for this grammar is

	int	+	*	()	\$
exp	$P1$			$P1$		
exp'		$P2$			$P3$	$P3$
$term$	$P4$			$P4$		
$term'$		$P6$	$P5$		$P6$	$P6$
$factor$	$P7$			$P8$		

Example (Cont.)

Consider parsing the symbol string

int + int

Stack	Input

How to Construct a Predictive Parser

To construct a predictive table parser we need two functions $FIRST(X)$ and $FOLLOW(X)$.

For a string (or symbol) X , $FIRST(X)$ is the set of all terminal symbols with which strings derived from X can start (plus ϵ if X can be reduced to ϵ).

$FOLLOW(X)$ is the set of all terminal symbols that can immediately follow X in some sentence (plus $\$$ if X can be the last symbol in some sentence).

We can then apply the following algorithm to compute the parsing table

For each production $A \rightarrow w$ do the following:

- For each terminal a in $FIRST(w)$,
add $A \rightarrow w$ to $M[A, a]$.
- If $\epsilon \in FIRST(w)$ then
for each terminal or $\$$ symbol $b \in FOLLOW(A)$ then
add $A \rightarrow w$ to $M[A, b]$.
- If $\epsilon \in FIRST(w)$ and $\$$ in $FOLLOW(A)$ then
add $A \rightarrow w$ to $M[A, \$]$.

Add error entries to all other cells in $M[A, t]$.

FIRST

For any sentence w , $FIRST(w)$ is the set of terminals that begin the strings derived from w . If ϵ can be derived from α , then $\epsilon \in FIRST(w)$.

For example, consider the grammar:

$$\begin{aligned} S &\rightarrow B C D \\ B &\rightarrow \epsilon \\ B &\rightarrow b b B \\ C &\rightarrow \epsilon \\ C &\rightarrow c C \\ D &\rightarrow d \\ D &\rightarrow d D \end{aligned}$$

where S , B , C and D are non-terminals and b , c and d are terminals.

We have that

$$\begin{aligned} FIRST(b) &= \\ FIRST(c) &= \\ FIRST(d) &= \\ FIRST(d D) &= \\ FIRST(D) &= \\ FIRST(c C) &= \\ FIRST(C) &= \\ FIRST(b b B) &= \\ FIRST(B) &= \\ FIRST(B C D) &= \\ FIRST(S) &= \end{aligned}$$

Computing FIRST

For a string $\alpha = Y_1, \dots, Y_k$, we can compute $FIRST(\alpha)$ as follows:

- place ϵ in $FIRST(X)$ if for all $i = 1, \dots, k$, ϵ is in $FIRST(Y_i)$;
- for $i = 1, \dots, k$, place a in $FIRST(X)$ if ϵ is in $FIRST(Y_1), \dots, FIRST(Y_i)$ and $a \in (FIRST(Y_{i+1}) \setminus \{\epsilon\})$.

We can compute $FIRST(X)$ for any grammar symbol X by exhaustively applying the following rules:

- If X is a terminal, $FIRST(X)$ is $\{X\}$.
- If $X \rightarrow \epsilon$ is a production, add ϵ to $FIRST(X)$.
- If $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, then add each element $b \in FIRST(Y_1 Y_2 \dots Y_k)$ to $FIRST(X)$.

Computing FIRST Example

For the grammar:

$S \rightarrow B C D$
 $B \rightarrow \epsilon$
 $B \rightarrow b b B$
 $C \rightarrow \epsilon$
 $C \rightarrow c C$
 $D \rightarrow d$
 $D \rightarrow d D$

We have that

$FIRST(b) =$
 $FIRST(c) =$
 $FIRST(d) =$
 $FIRST(B) =$
 $FIRST(C) =$
 $FIRST(D) =$
 $FIRST(S) =$

Computing FIRST (Cont.)

Consider the grammar:

$exp \rightarrow term \ exp'$
 $exp' \rightarrow +exp$
 $exp' \rightarrow \epsilon$
 $term \rightarrow factor \ term'$
 $term' \rightarrow *term$
 $term' \rightarrow \epsilon$
 $factor \rightarrow \mathbf{int}$
 $factor \rightarrow (exp)$

Exercise: Compute

symbol	FIRST
+	
int	
(
)	
exp	
exp'	
term	
term'	
factor	

FOLLOW

For any non-terminal A , $FOLLOW(A)$ is the set of terminals that can immediately follow A in the strings derived from the start symbol S . If A can be the last symbol in some sentence, then $\$ \in FOLLOW(A)$.

For example, recall the grammar:

$S \rightarrow B \ C \ D$
 $B \rightarrow \epsilon$
 $B \rightarrow b \ b \ B$
 $C \rightarrow \epsilon$
 $C \rightarrow c \ C$
 $D \rightarrow d$
 $D \rightarrow d \ D$

We have that

$FOLLOW(B) =$
 $FOLLOW(C) =$
 $FOLLOW(D) =$
 $FOLLOW(S) =$

What happens if we add the rule

$D \rightarrow \epsilon$

Computing FOLLOW

We can compute $FOLLOW(A)$ by exhaustively applying the following rules:

- Place \$ in $FOLLOW(S)$.
- If $A \rightarrow \alpha B \beta$ is a production, then everything in $FIRST(\beta)$ except for ϵ is placed in $FOLLOW(B)$.
- If there is a production $A \rightarrow \alpha B$, then everything in $FOLLOW(A)$ is placed in $FOLLOW(B)$.
- If there is a production $A \rightarrow \alpha B \beta$ where $\epsilon \in FIRST(\beta)$, then everything in $FOLLOW(A)$ is placed in $FOLLOW(B)$.

For the grammar:

$$\begin{aligned} S &\rightarrow B C D \\ B &\rightarrow \epsilon \\ B &\rightarrow b b B \\ C &\rightarrow \epsilon \\ C &\rightarrow c C \\ D &\rightarrow d \\ C &\rightarrow d D \end{aligned}$$

We compute FOLLOW as follows

$$\begin{aligned} FOLLOW(B) &= \\ FOLLOW(C) &= \\ FOLLOW(D) &= \\ FOLLOW(S) &= \end{aligned}$$

Computing FOLLOW (Cont.)

Consider the grammar:

$$\begin{aligned} exp &\rightarrow term\ exp' \\ exp' &\rightarrow +exp \\ exp' &\rightarrow \epsilon \\ term &\rightarrow factor\ term' \\ term' &\rightarrow *term \\ term' &\rightarrow \epsilon \\ factor &\rightarrow \mathbf{int} \\ factor &\rightarrow (exp) \end{aligned}$$

Exercise: Compute

symbol	FOLLOW
exp	
exp'	
$term$	
$term'$	
$factor$	

Table Construction Example

Recall the construction algorithm:

For each production $A \rightarrow w$ do the following:

- For each terminal a in $FIRST(w)$, add $A \rightarrow w$ to $M[A, a]$.
- If $\epsilon \in FIRST(w)$ then for each terminal or $\$$ symbol $b \in FOLLOW(A)$ then add $A \rightarrow w$ to $M[A, b]$.
- If $\epsilon \in FIRST(w)$ and $\$$ in $FOLLOW(A)$ then add $A \rightarrow w$ to $M[A, \$]$.

Add error entries to all other cells in $M[A, t]$.

Recall

exp	\rightarrow	$term$	exp'	(P1)
exp'	\rightarrow	$+exp$		(P2)
exp'	\rightarrow	ϵ		(P3)
$term$	\rightarrow	$factor$	$term'$	(P4)
$term'$	\rightarrow	$*term$		(P5)
$term'$	\rightarrow	ϵ		(P6)
$factor$	\rightarrow	int		(P7)
$factor$	\rightarrow	(exp)		(P8)

We have that

$$FIRST(exp) = FIRST(term) = FIRST(factor) = \{(\text{int})\}$$

$$FIRST(exp') = \{+, \epsilon\}$$

$$FIRST(term') = \{*, \epsilon\}$$

$$FOLLOW(exp) = FOLLOW(exp') = \{\}, \$$$

$$FOLLOW(term) = FOLLOW(term') = \{+, \}, \$$$

$$FOLLOW(factor) = \{+, *, \}, \$$$

The parsing table for this grammar is

	int	+	*	()	\$
exp						
exp'						
$term$						
$term'$						
$factor$						

Left Recursion Revisited

Left-recursive grammars cannot be processed with predictive table parsers either.

This can even be shown without knowing the precise parsing table.

Assume G contains the left-recursive production $E \rightarrow E + E \mid id$

Consider the parser state:

- top of stack = E ,
- current input = id ,
- $E \rightarrow E + E \in M[E, id]$

The predictive table parser would execute the following actions:

- $\text{pop}()$;
- $\text{push}(E)$; $\text{push}(+)$; $\text{push}(E)$;
- no advance in lookahead

Thus neither the top of the stack nor the current input symbol change and the parser will loop infinitely.

LL(1) Grammars

As we have seen not all grammars are suitable for predictive parsing since they cannot effectively predict which production should be applied.

This idea of “being suitable” for predictive parsing is captured in the $LL(k)$ property for a grammar. A grammar is called $LL(k)$ if the production to be applied can be determined with a maximum lookahead of k symbols. We are particularly interested in $LL(1)$.

A grammar is $LL(1)$ if the generated predictive-parse table has no cells with multiple entries.

THEOREM: A grammar G is $LL(1)$ iff for each non-terminal A :

- If $A \rightarrow \alpha$ and $A \rightarrow \beta$ are different productions in G , then $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$.

(Note at most one of α and β can derive the empty string).

- If $\beta \Rightarrow^* \epsilon$,

$$FIRST(\alpha) \cap FOLLOW(A) = \emptyset.$$

If a grammar is not $LL(1)$ you can sometimes use left recursion removal and left factoring to transform it into an equivalent $LL(1)$ grammar.

However, not all context-free grammars have an equivalent $LL(1)$ grammar.

Error Correction

Sensible error handling is extremely important: imagine if your compiler simply aborted the compilation of a faulty program with just the message “error - compilation aborted”!

In fact the usual case for program compilation is that there are syntax errors in the program!

Four levels of behaviour are possible when an error is encountered:

1. locate & report (without further analysis),
2. diagnose (report the type of error),
3. recover (i.e. skip over the error and continue parsing afterwards),
4. correct.

Any good compiler implements some level of recovery. Correction is the “holy grail” of parsing, but only possible in special cases.

Error Correction in Table-Driven Predictive Parsers

The stack in the table-driven recursive parser makes explicit the terminals and non-terminals it expects to match with the rest of the input. This can be used to guide error recovery.

A **syntax error** is detected when

- the terminal on top of the stack does not match the input token, or
- the non-terminal symbol on top of the stack A has an empty entry $M[A, a]$ for the current input symbol a .

Error Correction in Table-Driven Predictive Parsers (Cont.)

Panic-mode error recovery is based on the idea of skipping input symbols until a token in a selected set of synchronizing tokens appears.

One heuristic is to place all of the symbols in $FOLLOW(A)$ into the synchronizing set for non-terminal A .

We then skip input until a member of $FOLLOW(A)$ is encountered, and pop A from the stack.

Another is to place all of the symbols in $FIRST(A)$ into the synchronizing set for non-terminal A .

We then skip input until a member of $FIRST(A)$ is encountered.

See Aho et al Section 4.4 for more details.

Error Correction – Example

Recall

$exp \rightarrow term\ exp' \quad (P1)$
 $exp' \rightarrow +exp \quad (P2)$
 $exp' \rightarrow \epsilon \quad (P3)$
 $term \rightarrow factor\ term' \quad (P4)$
 $term' \rightarrow *term \quad (P5)$
 $term' \rightarrow \epsilon \quad (P6)$
 $factor \rightarrow int \quad (P7)$
 $factor \rightarrow (exp) \quad (P8)$

The parsing table with error correction for this grammar is

	int	+	*	()	\$	first	follow
exp	$P1$	$skip$	$skip$	$P1$	$synch$	$synch$	$(\ int$	$)\ \$$
exp'	$skip$	$P2$	$skip$	$skip$	$P3$	$P3$	$+$	$)\ \$$
$term$	$P4$	$synch$	$skip$	$P4$	$synch$	$synch$	$(\ int$	$+)\ \$$
$term'$	$skip$	$P6$	$P5$	$skip$	$P6$	$P6$	$*$	$+)\ \$$
$factor$	$P7$	$synch$	$synch$	$P8$	$synch$	$synch$	$(\ int$	$+\ *)\ \$$

A *skip* means to skip the current input terminal and continue.

A *synch* does the following where A is the non-terminal on top of the stack:

- skip input terminals until a symbol in $FIRST(A)$ or $FOLLOW(A)$ is encountered
- if the symbol is in $FOLLOW(A)$ pop A off the stack
- continue.

Error Correction – Example (Cont.)

Consider parsing the symbol string

)int + *int

Summary

We have looked at:

- Table-driven predictive parsing.
- LL(1) grammars.

Homework

- Read Section 4.4 of Aho et al.
- Give the predictive parsing table for the grammar

$$\begin{aligned} S &\rightarrow B C D \\ B &\rightarrow \epsilon \\ B &\rightarrow b b B \\ C &\rightarrow \epsilon \\ C &\rightarrow c C \\ D &\rightarrow \epsilon \\ D &\rightarrow d D \end{aligned}$$

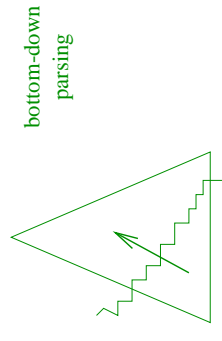
and use your table (with table-driven predictive parsing) to determine if $b c$ and $d c$ are in the language of the grammar.

- Modify the grammar for arithmetic expressions to include division and subtraction.
- Give the predictive parsing table for this extended grammar.

Programming Language Implementation V

Bottom-Up Parsing

In **bottom-up parsing** the parse tree is built from the sentence upwards using the productions in reverse until the start symbol is discovered.



In this lecture we will look at **bottom-up parsing**.

- **Table driven bottom-up parsing.**
- **LR parsing.**
- **bison.**

The material is (loosely) based on Aho et al Chapter 4.

In the next three lectures we will look at two approaches to **bottom-up parsing**: **table-driven bottom-up parsers** and a general **dynamic programming**.

Why Bottom-Up Parsing?

- Expressiveness
 - The class LL(1) of grammars that can be parsed deterministically using a top-down parser is a proper subset of the class LR(1) of grammars that can be parsed deterministically using bottom-up techniques with a lookahead of 1.
 - In particular top-down parsers loop infinitely for left-recursive grammars while bottom-up parser can process left-recursion.
 - LR parsing, the most general known non-backtracking shift-reduce method, works for almost all the known programming language constructs (in part because languages are designed so they can be parsed with LR parsers).
- Error Handling
 - Several well-known error handling techniques are applicable for bottom-up parsing. LR parsing can detect errors as soon as it is possible on a left to right scan.
- Disadvantage: Table construction is expensive and relatively complex.

Shift Reduce Parsing

Most efficient bottom-up parsing algorithms are based on shift-reduce parsing. This

- processes symbols left-to-right
- has limited lookahead
- no backtracking
- constructs the derivation tree bottom-up.

Operator-precedence parsing and LR parsing are well-known examples of shift-reduce based parsing.

Shift-reduce parsing constructs a **rightmost derivation in reverse**, reducing a **handle** at each step.

A **handle** h in a sequence of grammar symbols s is a subsequence that matches the RHS of some production $P: A \rightarrow h$, and whose reduction to A , the LHS of P , is a step along a reverse rightmost derivation of s .

Shift Reduce Parsing

Consider the grammar

$$exp \rightarrow exp + exp \mid exp * exp \mid int \mid (exp)$$

This has the **rightmost derivation**

$$\begin{aligned} exp &\rightarrow_{rm} \underline{exp} + exp \\ &\rightarrow_{rm} \underline{exp} + \underline{exp} * exp \\ &\rightarrow_{rm} \underline{exp} + \underline{exp} * \underline{int}_3 \\ &\rightarrow_{rm} \underline{int}_1 + \underline{int}_2 * \underline{int}_3 \\ &\rightarrow_{rm} \underline{int}_1 + \underline{int}_2 * int_3 \end{aligned}$$

Shift-reduce parsing works as follows:

Right-sentential form	Production
<u>int</u> ₁ + int ₂ * int ₃	exp → int
exp + <u>int</u> ₂ * int ₃	exp → int
exp + exp * <u>int</u> ₃	exp → int
exp + <u>exp</u> * exp	exp → exp * exp
<u>exp</u> + exp	exp → exp + exp
exp	

where the underlined symbols are called **handles**.

Note that this grammar is ambiguous and $exp + exp * exp$ has two possible handles.

Stack Implementatation of Shift-Reduce Parsing

The idea is to use a **stack** to hold the grammar symbols (terminals and non-terminals).

The parser starts with

STACK INPUT
\$ a₁a₂...a_n\$

and wants to reach

STACK INPUT
\$ S \$

The parser repeatedly performs one of the following actions

- A **shift** action which pushes the next input symbol on top of the stack.
- A **reduce** action which pops a **handle** from the stack chooses a production and pushes the production's LHS symbol onto the stack.
- An **accept** action when the string is parsed.
- An **error** action when the parser discovers a syntax error.

Example of Shift-Reduce Parsing

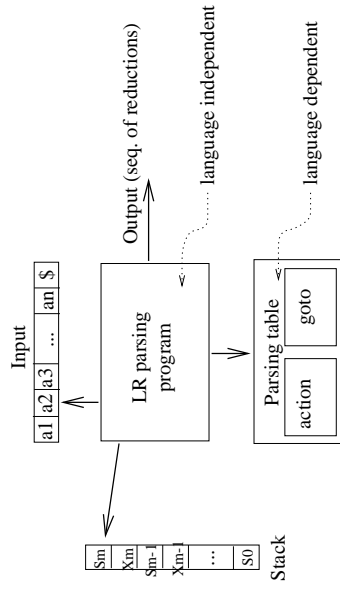
STACK	INPUT	ACTION
\$	$int_1 + int_2 * int_3$ \$	shift
\$ int_1	$+int_2 * int_3$ \$	reduce by $exp \rightarrow int$
\$ exp	$+int_2 * int_3$ \$	shift
\$ $exp +$	$int_2 * int_3$ \$	shift
\$ $exp + int_2$	$*int_3$ \$	reduce by $exp \rightarrow int$
\$ $exp + exp$	$*int_3$ \$	shift
\$ $exp + exp *$	int_3 \$	shift
\$ $exp + exp * int_3$	\$	reduce by $exp \rightarrow int$
\$ $exp + exp * exp$	\$	reduce by $exp \rightarrow exp * exp$
\$ $exp + exp$	\$	reduce by $exp \rightarrow exp + exp$
\$ exp	\$	accept

Shift-reducing parsing is based on the fact that a handle must always appear on top of the stack, never hidden inside.

LR Parsers

LR parsers

- work for most context-free languages,
- can be implemented efficiently,
- allow good error handling,
- are very complex but parser generators help.



LR Parsing Algorithm

```

repeat forever
   $s := top(stack)$ 
   $a :=$  current input symbol
  if  $action[s, a]$  is shift  $s'$  then
    push  $a$  then  $s'$  on to stack
  advance to next input symbol
  else if  $action[s, a]$  is reduce  $A \rightarrow \beta$  then
    pop  $2 \times |\beta|$  symbols off stack
     $s' := top(stack)$ 
    push  $A$  then  $goto[s', A]$  on to stack
    output " $A \rightarrow \beta$ "
  else if  $action[s, a]$  is accept then
    return
  else error()

```

107

Example of LR Parsing

Recall the grammar

```

exp → exp + term (1)
exp → term (2)
term → term * factor (3)
term → factor (4)
factor → (exp) (5)
factor → int (6)

```

The parsing table for this grammar is

STATE	ACTION				GOTO			
	int	+	()	\$	exp	term	factor
0	s5		s4			1	2	3
1		s6			acc			
2		r2	s7	r2	r2			
3		r4	r4	r4	r4			
4		s5		s4		8	2	3
5		r6	r6	r6	r6			
6		s5		s4			9	3
7		s5		s4				10
8		s6		s11				
9		r1	s7	r1	r1			
10		r3	r3	r3	r3			
11		r5	r5	r5	r5			

where

si is shift and stack state i

rj is reduce using production j

acc is accept.

108

Example of LR Parsing (Cont.)

Parser Generators – bison

Creating LR parsers by hand is quite tedious and error prone. Instead we can use a parser generator like yacc or bison.

bison is part of the Open Software Foundation's GNU system.

bison provides an interface to C so that the programmer can specify C code to be executed whenever a particular reduction takes place. Each symbol in the grammar has a single attribute which this can compute.

bison takes an LALR(1) grammar plus C annotation, `gram.y`, and produces a C file `gram.tab.c` which contains the parsing function `yyparse`.

bison assumes that the scanner is called `yylex` which may be generated by flex or written by hand.

STACK	INPUT	ACTION
0	$int_1 + int_2 \$$	

Bison

A bison specification is similar to that of flex. It has the form

```
%{  
<C declarations>  
}%  
<Bison declarations>  
%%  
<Grammar rules>  
%%  
<The user's C functions>
```

The **C declarations** section introduces types and variables which are used in the semantic rules. Macro definitions and includes are also placed here.

The **Bison declarations** list the tokens (ie terminal symbols) and may assign individual types to symbol attributes. Precedences and associativity rules for operators may be placed here.

The **grammar rules** list the productions and the associated semantic actions.

The **user code** is copied to `gram.tab.c`.

Simple Bison Example

Consider the “silly” grammar

$$\begin{aligned} S &\rightarrow B C D \\ B &\rightarrow \epsilon \\ B &\rightarrow b b B \\ C &\rightarrow \epsilon \\ C &\rightarrow c C \\ D &\rightarrow d \\ C &\rightarrow d D \end{aligned}$$

The following bison specification `silly.y` recognises sentences in the grammar

```
/* No declarations */  
%% /* Grammar rules */  
  
S : B C D;  
B : /* empty */  
  | 'b' 'b' B  
  ;  
C : /* empty */  
  | 'c' C  
  ;  
D : 'd'  
  | 'd' D  
  ;  
%% /* Auxiliary C code */  
  
main(){  
  yyparse(); }  
  
yyperror(char *s){ /* called by yyparse on error */  
  printf("%s\n",s); }
```

Another Bison Example

The following bison specification `calc.y` defines a simple calculator.

```
/* Declarations */
%{
#define YYSTYPE int
%}

%token NUM
%left '+' '-'
%left '*' '/'
%% /* Grammar rules */

line
: /* empty */
| line exp '\n' { printf("%d\n", $2); }
;

exp
: exp '+' exp { $$ = $1 + $3; }
| exp '-' exp { $$ = $1 - $3; }
| exp '*' exp { $$ = $1 * $3; }
| exp '/' exp { $$ = $1 / $3; }
| '(' exp ')' { $$ = $2; }
| NUM /* $$ = $1; */
;

%% /* Auxiliary C code */

main() {
    yyparse();
}

yyerror(char *s) { /* called by yyparse on error */
    printf("%s\n", s);
}
```

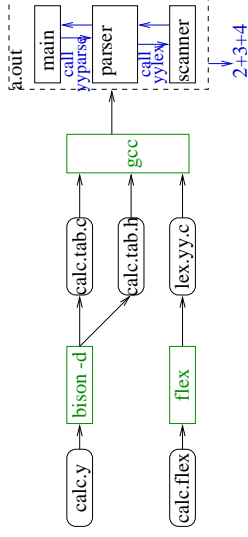
Using Bison with Flex

The `-d` option to `bison` causes it to output a header file `gram.tab.h` detailing the tokens for use in the flex file.

For example the flex file for our application, `calc.flex`, is

```
%{
#include "calc.tab.h"
%}
DIGIT [0-9]
WSPACE [ \t]
%%
{DIGIT}+ { yylval = atoi(yytext); return NUM; }
{-WSPACE}+
\n {return yytext[0];}
. {return yytext[0];}
```

Using Bison with Flex (Cont.)



We compile the whole application as follows:

```

% bison -d calc.y
% ls
% calc.tab.c calc.tab.h calc.y calc.flex
% flex calc.flex
% gcc calc.tab.c lex.yy.c -lfl
% a.out
(54 * 63) + 102 / 5
3422
2 + 3 + 4
9
5 + *4
parse error

```

Error Handling

By default a parser generated by `bison` will call the routine `yyerror` when it finds a syntax error and stop parsing.

However it also supports better error handling. There is a predefined symbol `error`. This may occur anywhere in the RHS of a production and will match the syntactically invalid sub-string. Parsing will continue and a call to `yyerror` indicates that error handling is complete.

For example in the desk calculator we might decide to skip lines with a syntax error

```

line
: /* empty */
| line exp '\n' { printf("%d\n", $2); }
| line error '\n' { yyerror; }
;

```

Now the calculator continues after finding an error

```

% a.out
2 + 3 + 4
9
5 + *4
parse error
5 + 4
9

```

Summary

We have looked at bottom-up parsing:

- Table driven bottom-up parsing.
- LR parsing.
- `bison`.

Homework

- Read Chapter 4 of Aho et al.
- Get the online `bison` documentation.
- Write a flex file for the silly grammar and generate the `bison` based parser.
- Generate the `bison` based desk calculator.
- Extend the desk calculator to support real numbers, `sqrt`, `sin` and `cos` by modifying the `bison` and flex specification.

Programming Language Implementation VI

In this lecture we will continue to look at bottom-up parsing.

- Constructing the LR parsing table.

The material is (loosely) based on Aho et al Chapter 4.

LR Parsers

In the last lecture we met LR parsers. In this lecture we will learn how to construct the LR parsing table.

There are three well-known methods:

- **Simple LR (SLR)**: this is the simplest and uses limited lookahead in the table construction.
- **Canonical LR**: this is the most powerful but leads to large parsing tables.
- **LALR**: is slightly less powerful than Canonical LR but leads to smaller parsing tables and suffices for almost all programming language constructs. It is used in `yacc` and `bison`.

We will focus on SLR parsing table construction since it is the easiest to understand and forms the basis for the other two methods.

Viable Prefixes

The basic idea behind all LR parser methods is to recognize handles of the grammar. They do this by executing a DFA that recognizes viable prefixes of the grammar.

If $S \Rightarrow^*_{rrm} uXw \Rightarrow^*_{rrm} uvw$ then v is a handle of uvw and each prefix of uv is a viable prefix.

The key idea in LR parser construction is to construct from the grammar a DFA to recognize when a handle has been found and which production to use. The construction method is similar to how we turn a NFA into a DFA.

Items

Construction of the parsing table requires us to keep track of “partially evaluated” grammar rules. **Items** formalize this notion.

An **(LR(0)) item of a grammar is a production with a dot somewhere in the RHS.**

An item indicates how much of a production we have seen at a particular point in the parsing process. It means that we have seen the symbols to the left of the “.” and expect to see those symbols to the right.

For example, production $exp \rightarrow exp + exp$ yields 4 items:

$exp \rightarrow \cdot exp + exp$
 $exp \rightarrow exp \cdot + exp$
 $exp \rightarrow exp + \cdot exp$
 $exp \rightarrow exp + exp \cdot$

The production $exp \rightarrow \epsilon$ generates only the single item

$exp \rightarrow \cdot$

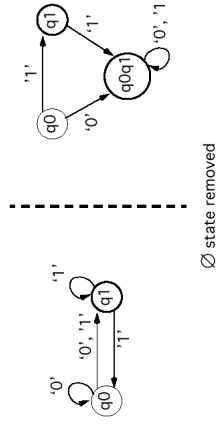
The idea is to group the items of a grammar into sets that form the states of a DFA that recognizes the viable prefixes of the grammar.

This grouping is essentially an NFA - ϵ -DFA subset conversion, and the states of the corresponding NFA correspond to single items.

Review of NFA to DFA Conversion

A finite state automata is **deterministic** if it has no transitions via ϵ and at most one successor state for each pair (q, a) where $q \in Q$ and $a \in \Sigma$. We call such an automata a **DFA**.

Every **NFA** has a corresponding **DFA** which recognizes the same language. This can be generated using the **subset construction algorithm**:



The **DFA** has states corresponding to **sets of states in the original NFA**.

We start from the new “super” start state which is the old start state + all states which are ϵ successors.

The successor state of a super state S for character a is the super state containing all states which can be reached from a state in S under a and adding their ϵ successor states.

Any super state which contains a final state is a final state.

LR construction

To construct the viable prefix recognizing DFA we need the so-called **canonical LR(0) collection**, which will be the set of states of this automaton.

We will make use of two functions defined on items:

- **closure**
- **goto**.

For simplicity we will **augment** the initial grammar with a new start symbol S' and a new production $S' \rightarrow S$ where S is the original start symbol.

We need this extra production to recognize the reduction that leads to acceptance.

Closure

The **closure** of a set of items I , written $\text{closure}(I)$, is computed by:

- Start with I .

- If

$$A \rightarrow \alpha \cdot B\beta$$

is in the set and

$$B \rightarrow \gamma$$

is a production, add

$$B \rightarrow \cdot \gamma$$

Continue doing this until nothing can be added.

The intuition is that if $A \rightarrow \alpha \cdot B\beta$ is in $\text{closure}(I)$ then we might expect to next see a substring derivable from $B\beta$. Since $B \rightarrow \gamma$ is a production we might therefore expect to see a string derivable from γ .

Consider the augmented grammar

```
exp'  → exp
exp   → exp + term
exp   → term
term  → term * factor
term  → factor
factor → (exp)
factor → int
```

If I is $\{exp' \rightarrow \cdot exp\}$ then what is $\text{closure}(I)$?

Valid Items

Next we have to construct the *goto* function. This function (together with the item collection) embodies the idea of a **valid item**.

An item $X \rightarrow YZ$ is valid for a viable prefix vY if there is a rightmost derivation $S \Rightarrow_{rrm}^* vXw \Rightarrow_{rrm} vYZw$.

Informally this means: If we find vY on top of the stack and $X \rightarrow YZ$ is a valid item then if Z is not empty we have not yet shifted the handle on the stack so we must shift (to move YZ on the stack). If Z is empty then Y must be the handle so we have to reduce by $X \rightarrow YZ$.

(This holds provided that no conflicts occur).

Goto

Let I be a set of items and X a grammar symbol.

The set $goto(I, X)$ is the set of items we arrive at by processing the symbol X . This will be the state transition function of the automaton.

More precisely, I is the set of all valid items for some viable prefix w , and $goto(I, X)$ returns the set of all valid items for the viable prefix wX .

To do so we take each $A \rightarrow \alpha \cdot X\beta$ in I and produce

$$A \rightarrow \alpha X \cdot \beta$$

Now we take the closure of these.

Consider the grammar

$exp' \rightarrow exp$
 $exp \rightarrow exp + term$
 $exp \rightarrow term$
 $term \rightarrow term * factor$
 $term \rightarrow factor$
 $factor \rightarrow (exp)$
 $factor \rightarrow int$

If I is

$$\{exp' \rightarrow exp, exp \rightarrow exp \cdot + term\}$$

then $goto(I, +)$ is what?

Constructing the Sets of LR(0) items

To build the parsing table we first compute the collection of sets of LR(0) items, C , for the augmented grammar. These correspond to the states in the DFA.

- Initialize C to $\{closure(\{S' \rightarrow \cdot S\})\}$.
- repeat
 - for each set $I \in C$ and each grammar symbol X do
 - if $goto(I, X)$ is not empty then
 - $C := C \cup \{goto(I, X)\}$
- until C is unchanged

For the grammar

```

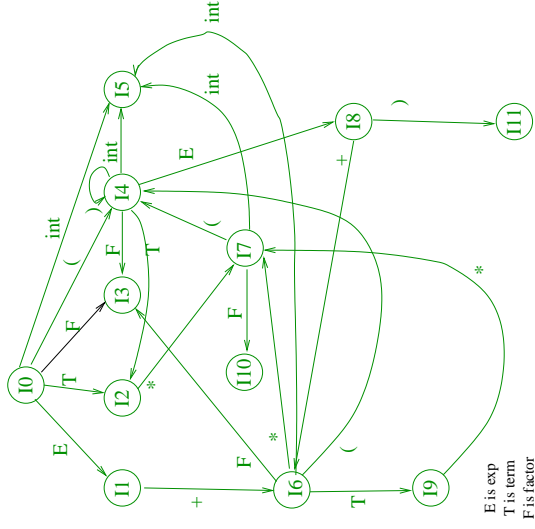
exp'  → exp
exp   → exp + term
exp   → term
term  → term * factor
term  → factor
factor → (exp)
factor → int
  
```

we have the collection of sets of items:

```

I0: exp' → ·exp
      exp  → ·exp + term
      exp  → ·term
      term → ·term * factor
      term → ·factor
      factor → ·(exp)
      factor → ·int
I1: exp' → exp·
      exp  → exp · + term
      exp  → term·
I2: exp  → term·
      term → term · * factor
      term → factor·
I3: term → factor·
I4: factor → (·exp)
      exp  → ·exp + term
      exp  → ·term
      term → ·term * factor
      term → ·factor
      factor → ·(exp)
      factor → ·int
I5: factor → (exp)·
      exp  → exp + term
      exp  → term * factor
      exp  → ·factor
      factor → ·(exp)
      factor → ·int
I6: factor → (exp)·
      exp  → exp + term
      exp  → term * factor
      exp  → ·factor
      factor → ·(exp)
      factor → ·int
I7: factor → (exp)·
      exp  → exp + term
      exp  → term * factor
      exp  → ·factor
      factor → ·(exp)
      factor → ·int
I8: factor → (exp)·
      exp  → exp + term
      exp  → term * factor
      exp  → ·factor
      factor → ·(exp)
      factor → ·int
I9: factor → (exp)·
      exp  → exp + term
      exp  → term * factor
      exp  → ·factor
      factor → ·(exp)
      factor → ·int
I10: factor → (exp)·
      exp  → exp + term
      exp  → term * factor
      exp  → ·factor
      factor → ·(exp)
      factor → ·int
I11: factor → (exp)·
      exp  → exp + term
      exp  → term * factor
      exp  → ·factor
      factor → ·(exp)
      factor → ·int
  
```

The Characteristic Automaton



Essentially we have constructed the DFA above. Its states represent the canonical LR(0) item collection and the state transition function is derived from the *goto* function from above.

Each state is an end state and the state I_0 (which contains the item $S' \rightarrow \cdot S$) is its initial state.

If this DFA (starting in state I_0) processes some viable prefix v it reaches a state I_n that represents exactly the valid items for v .

Constructing the SLR Parsing Table

- Compute $C = \{I_0, I_1, \dots, I_n\}$ the collection of sets of LR(0) items for the augmented grammar.
- Make a state s_i for each I_i :
 1. If $A \rightarrow \alpha \cdot a\beta$ is in I_i and $goto(I_i, a) = I_j$ then set $action[i, a]$ to *shift* s_j . Note that a must be a terminal.
 2. If $A \rightarrow \alpha \cdot$ is in I_i then set $action[i, a]$ to *reduce* $A \rightarrow \alpha$ for all $a \in FOLLOW(A)$. Note that A may not be S' .
 3. If $S' \rightarrow S \cdot$ is in I_i then set $action[i, \$]$ to *accept*.
 4. For each nonterminal A , if $goto(I_i, A) = I_j$ then set $goto[s_i, A]$ to j .
- All other entries are set to *error*.

The initial parser state is the state corresponding to the item set I_i that contains $S' \rightarrow S$.

Constructing the SLR Parsing Table (Cont.)

What is the SLR parsing table for our example grammar?

- $exp \rightarrow exp + term$ (1)
- $exp \rightarrow term$ (2)
- $term \rightarrow term * factor$ (3)
- $term \rightarrow factor$ (4)
- $factor \rightarrow (exp)$ (5)
- $factor \rightarrow int$ (6)

SLR(1) Table Example

The LR parsing table for this grammar is

STATE	ACTION				GOTO				
	int	+	*	()	\$	exp	term	factor
0	s5			s4			1	2	3
1		s6			acc				
2		r2	s7		r2				
3		r4	r4		r4				
4	s5			s4			8	2	3
5		r6	r6		r6				
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1				
10		r3	r3		r3				
11		r5	r5		r5				

where

- si is shift and stack state i
- rj is reduce using production j
- acc is accept.

We have seen how the canonical item collection is constructed.

I_0 generates $action_{[s_0, ()} (= shift(s_4))$ and $action_{[s_0, id]} = shift(s_5)$.

I_1 generates $action_{[s_1, \$]} = accept$ and $action_{[1, +]} = shift(s_6)$.

I_2 generates $action_{[s_2, \$]} = action_{[s_2, +]} = action_{[s_2,)]} = reduce(E \rightarrow$

$T)$ since $follow(E) = \{ \$, +, \}$ and $action_{[2, *]} = shift(s_7)$.

etc.

SLR(1) Grammars

Steps (1)–(3) may lead to conflicting actions in which case the grammar is not **SLR(1)**.

The following grammar is an example of an unambiguous grammar which is **not SLR(1)**.

$$\begin{aligned} S &\rightarrow L = R \mid R \\ L &\rightarrow *R \mid \mathbf{id} \\ R &\rightarrow L \end{aligned}$$

The LR(0) collection is

$$\begin{aligned} I_0 : S' &\rightarrow \cdot S & I_4 : L &\rightarrow * \cdot R \\ S &\rightarrow \cdot L = R & R &\rightarrow \cdot L \\ S &\rightarrow \cdot R & L &\rightarrow \cdot * R \\ L &\rightarrow \cdot * R & L &\rightarrow \cdot \mathbf{id} \\ L &\rightarrow \cdot \mathbf{id} & & \\ R &\rightarrow \cdot L & I_5 : L &\rightarrow \mathbf{id} \cdot \\ \\ I_1 : S' &\rightarrow S \cdot & I_6 : S &\rightarrow L = \cdot R \\ & & R &\rightarrow \cdot L \\ I_2 : S &\rightarrow L \cdot = R & L &\rightarrow \cdot * R \\ R &\rightarrow L \cdot & L &\rightarrow \cdot \mathbf{id} \\ \\ I_3 : S &\rightarrow R \cdot & I_7 : L &\rightarrow * R \cdot \\ \\ I_8 : R &\rightarrow L \cdot & I_9 : S &\rightarrow L = R \cdot \end{aligned}$$

Consider I_2 and what $action[2, =]$ should be.

This is an example of a shift/reduce conflict.

The problem is that SLR(1) grammars are not powerful enough, performing a reduce whenever the next symbol is in the *FOLLOW* set even though contextual information might rule this out.

LALR(1) grammars work much like SLR but **remember** more about context. When setting up the parse table the LALR construction does not use the *FOLLOW* set but rather a subset of this. The grammar above is LALR(1).

Conflicts in Bison

Generation by `bison` may also lead to errors if the grammar is not LALR(1)

- **shift/reduce conflicts** in which case `bison` chooses to *shift*. Why?
- **reduce/reduce conflicts** usually indicate a serious problem.

Associativity and precedence declarations can sometimes overcome these problems.

Using `bison` with the `-v` flag will produce a file `gram.output` with information about such conflicts.

The tables generated are sparse, so usually a compact representation is used.

Summary

We have continued looked at bottom-up parsing:

- Constructing the LR parsing table.

Homework

- Read Chapter 4 of Aho et al.
- Construct the SLR parsing table for the grammar

$$\begin{aligned} S &\rightarrow B C D \\ B &\rightarrow \epsilon \\ B &\rightarrow b b B \\ C &\rightarrow \epsilon \\ C &\rightarrow c C \\ D &\rightarrow d \\ D &\rightarrow d D \end{aligned}$$

Programming Language Implementation VII

In this lecture we will finish looking at syntax analysis, i.e. parsing.

- **Generic bottom-up parsing – the CYK algorithm.**

We will also look at the answer to the 2nd assignment and the 3rd assignment.

Generic Bottom-Up Parsing

Not all grammars are LL(1) or even LALR(1).

Consider the grammar

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

This grammar is ambiguous: for instance consider *abab*. Therefore none of the techniques we have seen can be used for parsing.

Exercise: What language does this grammar generate?

There are several generic methods for parsing with context free grammars. One method based on **dynamic programming** is due to Cocke & Younger and Kasami (Often called the **CKY algorithm**). This works with any context free grammar, even ambiguous grammars.

The CKY algorithm can even be used for limited natural language parsing.

CKY Parsing

CKY Parsers require the grammar to be in Chomsky Normal Form (CNF).

This requires the grammar to be

- ϵ -free
- and each (non ϵ production) is of the form $A \rightarrow a$ or $A \rightarrow BC$ where a is a terminal and A, B, C are non-terminals.

A grammar is ϵ -free if it either has no ϵ -productions or there is exactly one ϵ -production $S \rightarrow \epsilon$ and the start symbol S does not occur on the right-hand side of any production.

Converting to Chomsky Normal Form

To convert a grammar into Chomsky Normal Form we

- make the grammar ϵ -free
- convert all productions into one of the forms $A \rightarrow a$ or $A \rightarrow BC$

Making a Grammar ϵ -free

To make a grammar with start symbol S ϵ -free

- Recursively determine those non-terminal symbols, X_1, \dots, X_n , which can generate ϵ , i.e. for which $X_i \Rightarrow^* \epsilon$.
- Repeat the following until no new production can be added:
for each production of form $X \rightarrow \alpha X_i \beta$ add the production $X \rightarrow \alpha \beta$.
- If there is a $S \rightarrow \epsilon$ production then introduce a new start symbol S' and add the productions $S' \rightarrow S$ and $S' \rightarrow \epsilon$. Remove all ϵ -productions where the LHS symbol is not S' .

Making a Grammar ϵ -free — Example

Consider the grammar

$$S \rightarrow aSbS \mid SaS \mid \epsilon$$

We can compute an equivalent ϵ -free grammar as follows

Converting Productions to $A \rightarrow a$ or $A \rightarrow BC$

It is easy to transform an ϵ -free grammar into Chomsky Normal Form. We simply add new non-terminal symbols for each terminal and for each production with more than two symbols in the RHS. Repeat the following process exhaustively:

Each production P that is not in the required form must be of the either of the forms $A \rightarrow a\alpha$ or $A \rightarrow B\alpha$ where a is a terminal, B a non-terminal and α a string over terminal and non-terminals.

- If P is of the form $A \rightarrow a\alpha$ replace P by $A \rightarrow XY$, where X, Y are fresh non-terminals. Add the production $X \rightarrow a$.
- If P is of the form $A \rightarrow B\alpha$ replace P by $A \rightarrow BY$, where Y is a fresh non-terminal.
- Add the production $Y \rightarrow \alpha$.

We have ignored the case $A \rightarrow B$. How is this handled?

Putting an Grammar in Chomsky Normal Form – Example

Our example grammar becomes:

The CKY Algorithm

Consider the input string $w = a_1 a_2 \dots a_n$.

We construct a $n \times n$ table T so that

$$T[i, j] = \{A \mid A \Rightarrow^* a_i a_{i+1} \dots a_j\}.$$

We do this diagonal by diagonal as follows:

- For $i = 1, \dots, n$, initialise $T[i, i]$ to $\{A \mid A \Rightarrow a_i\}$.
- Then we iteratively compute diagonals in terms of previous elements:

$$T[i, j] = \left\{ A \mid \begin{array}{l} A \rightarrow BC \text{ is a production and for some } k, i \leq k < j, \\ B \in T[i, k] \text{ and } C \in T[k+1, j] \end{array} \right\}$$

- w is in the language iff $S \in T[1, n]$ where S is the start symbol.

Simple Implementation of CYK

```
for j := 1 to n do
  T[j,j] := {A | A ⇒ a_j}
for i := j-1 to 1 do
  for k := i to j do
    if B ∈ T[i, k] and C ∈ T[k+1, j] and
       A → BC is a production then
      add A to T[i, j]
    end
  end
accept if S ∈ T[1, n].
end.
```

The CKY Algorithm -Example

Consider our example grammar and the string $abab$.

Summary

We have continued looked at bottom-up parsing:

- CYK algorithm.

And looked at the assignment.

Homework

- Give an algorithm to compute the non-terminals in a grammar which generate ϵ . [Hint: look at the algorithm for computing the FIRST set]

- Consider the grammar

$$exp \rightarrow exp + exp \mid int$$

Give a Chomsky Normal Form grammar for it. Now use the CKY Algorithm with the Chomsky Normal Form grammar to parse the string: $int_1 + int_2 + int_3$.

- Do the assignment!

What is the complexity of this method?