

## Programming Language Implementation

Bernd Meyer  
bernd.meyer@acm.org

Consultation: 4pm-6pm, Wednesdays  
Rm 148, Bldg 75

Basics of Programming Language Implementation.

- **Wk 8: Introduction, Lexical Analysis.**
- **Wk 9: Grammars, Top-down Parsing.**
- **Wk 10: Syntax-directed Translation.**
- **Wk 11: Bottom-up and LR(k) Parsing.**
- **Wk 12: Parser Generators, Code Generation.**
- **Wk 13: Revision Lectures for Exam.**

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

## References

- Well-established field with lots of good standard material.
  - *The classic*  
A.V. Aho, R. Sethi, J.D. Ullman  
*Compilers—Principles, Techniques and Tools*.  
Addison Wesley, 1986.
  - *A good, modern, very readable alternative, not as complete*  
K.D. Cooper and L. Torczon  
*Engineering a Compiler*.  
Morgan Kaufmann, 2004.
  - *For the connection to ML:*  
A.W. Appel  
*Modern Compiler Implementation in ML*.  
Cambridge University Press, 1997.
  - *Another alternative to the first two books emphasizing the implementation of different programming language paradigms:*  
A. Wilhelm and D. Maurer  
*Compiler Design*.  
Addison Wesley, 1995.

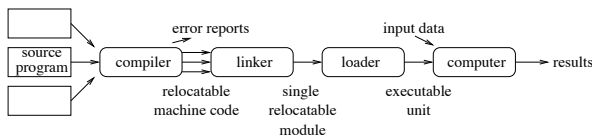
Much of the material in the lecture notes is based on Aho, Sethi, Ullman.

- For the underlying theory (formal languages and automata)  
J.E. Hopcroft and J.D. Ullman  
*Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.
- A useful source for compiler implementations (to obtain hands-on knowledge)  
<http://www.idiom.com/free-compilers/>

## Compiler Construction

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

- native machine code
- virtual machine code
- other programming language



- one of the best understood areas of computer science
- combines technical application with solid formal theory
- many industrial strength (de-facto standard) tools support the process

## Compiler vs. Interpreter

A compiler translates a program  $P_L$  written in language  $L$  into a program  $P_{LM}$  written in language  $LM$ . It is itself implemented in  $LC$ .

An Interpreter  $I$  executes a program  $P_L$  written in language  $L$  together with its input data  $D$  and is itself implemented in language  $LM$ .

	Compiler	Interpreter
Efficiency	+	
Static Checking	+	
Dynamic Code Change		+
Prototyping		+
Interactive Debugging		+

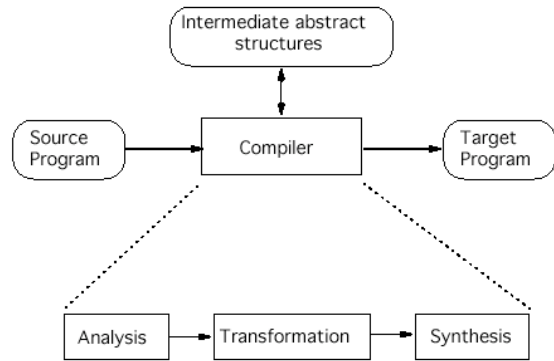
## Translators

Many techniques used in compiler construction are general translator techniques and have a much wider applicability.

- Preprocessors, Macro processors
- Text formatters
- HTML2TeX etc.
- Query Interpreters
- Silicon Compilers
- Postscript converters etc.
- TeX / LaTeX
- XML reader
- XSLT
- ...

All of these include translator components, some of these can be understood as programming languages (eg. Postscript, TeX )

## Structure of Compilation

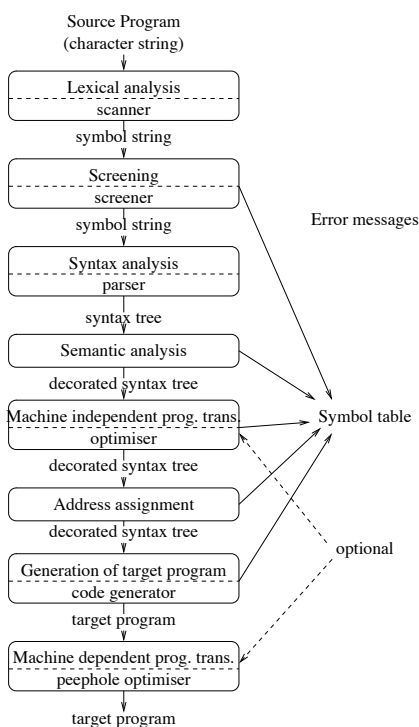


In reality the execution of these modules is not necessarily strictly sequenced.

**Single-pass compilers** perform all of these tasks interlinked with a single sweep over the source. This is only possible in special simple cases.

Normal compilers first construct an intermediate data structure (parse tree, see below) and traverse this for subsequent tasks.

## Phases of a Compiler



## Symbol Table

The central data structures are

- the **symbol table** which contains
  - identifiers  
*from screening*
  - tags for variables  
*from semantic analysis*
  - types for identifiers  
*from semantic analysis*
  - addresses for variables  
*from address allocation*
- the **parse tree**, which reflects the hierarchical program structure.

## Lexical Analysis

The **scanner** (lexical analyser) converts the source program (string of characters) into a sequence of tokens (i.e. lexical units).

Tokens are, for example

- identifiers (“this”, “x”, ...)
- numbers and other constants (“3.14”, “99”, “A String t”, ...)
- operators (“+”, “-”, “(”, ...)

At the same time meaningless whitespace (blanks, tabs, line breaks, etc.) are stripped.

In some cases the scanner already strips comments, too.

## 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 or regular (type 3) grammars

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

## Screening

Recall:

The Screener performs further analysis and classification of the tokens

This includes

- recognition of reserved words (“while”, “return” ...)
- stripping comments
- recognizing compiler directives (so-called pragmas)

Finally it creates the *symbol table* and links the tokens to the entries in the symbol table.

Scanner and Screener are in practice usually combined into a single module.

Modes of operation

- can run in a single pass before the syntax analysis
- can run interleaved with syntax analysis (on-demand)

## Screening

### Example

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

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

### Methods

As for the lexical analysis we use regular grammars (type 3 grammars) or regular expressions which are implemented with finite state automata (FSA).

Additional table look-up and calls to arbitrary (programmed) functions are used to achieve the additional capabilities, such as recognition of reserved words.

## Syntax Analysis

The task of the syntax analyser (called a parser) is to recover the hierarchical syntactical structure of the program which reflects its logical structure

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

Parsing is theoretically very well understood. The theory is based on context-free grammars (type 2 grammars) and thus pushdown automata, but it makes use of several refined classes of grammars.

## Error Handling

An important task of the syntax analyser is to check the syntactical correctness of a program and to locate, report (and diagnose) syntax errors.

Correct detection of (and recovery from) syntactical errors is one of the more difficult tasks in syntax analysis.

## Semantic Analysis

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

Typically determines the **kind** of each identifier.

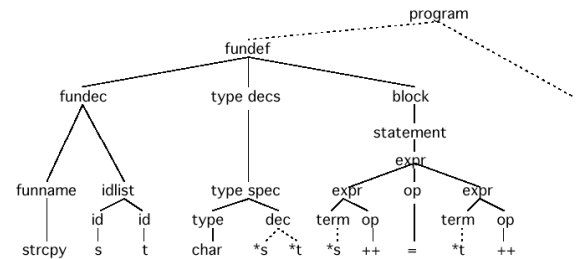
*Which identifiers are variables?*

Performs **type checking** and **type inference**.

Adds the kind and type information to the symbol table.

## Parse Tree

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



## 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:

- recognizing and Eliminating tail recursion
- recognizing and eliminating constant expressions
- combining expressions and eliminating intermediate variable assignments
- avoiding re-calculation of invariant parts in loops
- converting procedure/function/method calls to in-line code

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.

## Code Generation

Code generation produces the actual target program (machine instructions or VM instructions).

At this point the program must be error-free (tested by previous phases)

The central 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 at which point of the program to which register will each of these be assigned?  
Not all registers can perform all functions E.g. special registers used for multiplication. Results often returned in fixed register
- instruction sequencing

## 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 Optimization

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 code more efficient.

This is usually done by improving local segments of the code with

*“Peephole” optimization.*

The name refers to a small “peephole” passing over the code and revealing a local portion to be optimized in isolation.

Typical functions are

- changing instructions/instruction sequences to more efficient ones (special cases)
- removing unused instruction.

## Abstract Machines / Virtual Machines

Instead of compiling directly into machine code, the target is often a virtual machine.

A virtual machine is essentially an interpreter for a virtual language that runs on the target machine.

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.

As a consequence, translation into virtual machine code is easier. Native machine code is more complex, 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

- higher portability
- security (easier encapsulation)

However, native code is typically much faster.

## 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 and non-deterministic FSA
  - Equivalence and minimization of FSA
  - Pumping lemma
- **Scanner generators: MLLex, Flex.**

The material is (loosely) based on Appelt, Chapter 2 and Wilhelm and Maurer Chapter 7.

## Summary

We have looked at the main phases in a compiler

- the function of compilers and interpreters  
translation vs. execution
- the basic structure and processing phases of a compiler
  - lexical analysis
  - screening
  - parsing
  - semantic analysis
  - address assignment
  - code generation
  - optimization
- the central data structures used in a compiler  
symbol tables and parse trees

## 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.

## Lexical Analysis

Recall:

The **scanner** (lexical analyser) converts the source program (string of characters) into a sequence of tokens (i.e. lexical units).

Tokens are, for example

- identifiers (“this”, “x”, ...)
- numbers and other constants (“3.14”, “99”, “A String t”, ...)
- operators (“+”, “-”, “(”, ...)

At the same time meaningless whitespace (blanks, tabs, line breaks, etc.) are stripped.

In some cases the scanner already strips comments, too.

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

## Screening

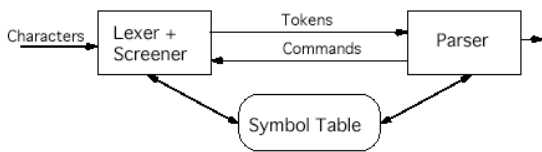
Recall:

The Screener performs further analysis and classification of the tokens

This includes

- recognition of reserved words (“while”, “return” ...)
- stripping comments
- recognizing compiler directives (so-called pragmas)

Finally it creates the *symbol table* and links the tokens to the entries in the symbol table.



## Words and Languages

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

We let  $\epsilon$  be the **empty word**, ie the null string.

A **(formal) language**  $L$  over  $\Sigma$  is a set of words over  $\Sigma$ .

We will use the following operations on formal languages:

$L_1 \cup L_2$		Union of languages
$L_1 L_2$	$\{xy \mid x \in L_1, y \in L_2\}$	Concatenation of languages
$L^n$	$\{x_1 \cdots x_n \mid x_i \in L, 1 \leq i \leq n\}$	
$L^*$	$\bigcup_{n \geq 0} L^n$	Closure of a language
$L^+$	$\bigcup_{n > 0} L^n$	
$\bar{L}$	$\Sigma^* - L$	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  $\Sigma$  are:

- $\epsilon$ , 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.

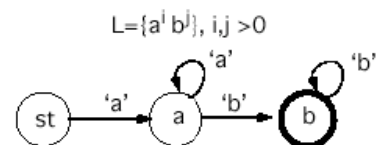
## Finite-State Automata

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**  $\Sigma$ ,
- 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.



### Transition Table

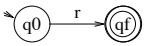
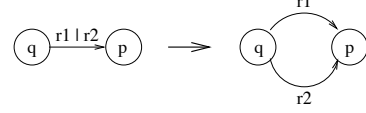
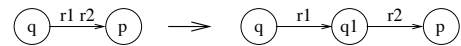
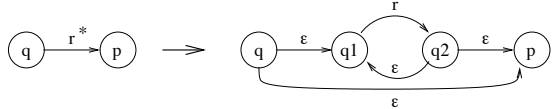
- An example NFA has
- input alphabet  $\{0, 1, \dots, 9, \dots, E\}$ ,
  - states  $\{0, \dots, 7\}$ ,
  - initial state 0,
  - final states  $\{1, 7\}$ , and
  - transition table,

	0, 1, ..., 9	.	E	$\epsilon$
0	{1, 2}	$\emptyset$	$\emptyset$	$\emptyset$
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	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

We usually represent an NFA using a **state transition diagram**. That for the previous NFA is:

### 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

- 1) 
- 2) 
- 3) 
- 4) 

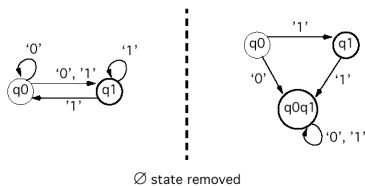
**Exercise:** What is a NFA for recognizing the language of  $a(a|b)^*$ ?

### NFA-DFA Equivalence

A finite state automata is **deterministic** if it has no transitions via  $\epsilon$  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 non-deterministic state automaton (NFA) can be transformed into an equivalent deterministic state automaton (DFA) such that both automata accept the same language.

The "trick" is to model all possible state combinations (in the NFA) explicitly as separate states (in the DFA). This is called the *subset construction*.



The obvious problem is that this leads to a combinatorial explosion 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 idea is to partition the original states into states which have **equivalent** accepting behaviour.

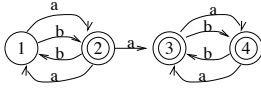
Initially the states are partitioned into the final states and the non-final states.

Repeatedly **split** a partition if its states lead to different accepting behavior for some character in the alphabet.

**Dead** (no path to final state) and **unreachable** (no path from start state) states are removed.

### Minimizing DFA Example

Find the minimal DFA for



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

### Regular Grammars

An alternative characterization of regular languages is via *regular grammars*. These are also called *type 3 grammars*.

A grammar is a quadruple  $G = (N, T, P, S)$  consisting of:

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

Such a grammar is a mechanism to construct words. Let  $x$  be a word over  $N \cup T$ . If it is possible to split  $x$  into  $uAv$  where  $u, v$  are words over  $N \cup T$  and  $A \in N$  such that  $(A \rightarrow q) \in P$ , we write

$$x \Rightarrow_G uqv$$

and say that  $uqv$  can be derived from  $x$  according to  $G$  (in a single step).

We write  $x_0 \Rightarrow_G^* x_n$  for  $x_0 \Rightarrow x_1 \Rightarrow \dots \Rightarrow x_n$ .

A grammar  $G = (N, T, P, S)$  generates (or accepts) the language  $L = \{x \mid S \Rightarrow_G^* x \wedge x \in T^*\}$ .

### Pumping Lemma

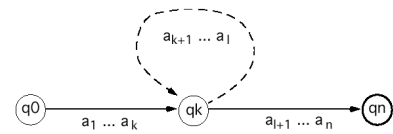
a fundamental result about regular languages is the *pumping lemma*.

It can be used to **prove that a language  $L$  is not regular**, i.e. that we cannot construct a DFA that accepts exactly this language.

#### Theorem: (Pumping Lemma)

Let  $L$  be some regular language. There is a finite constant  $n$  such that any word  $x \in L$  with  $\text{length}(x) \geq n$  can be split into three words  $u, v, w$  with  $x = uvw$  with  $\text{length}(uv) \leq n$  and  $\text{length}(v) > 0$  and  $\forall i > 0 : uv^iw \in L$ .

Consider a word in  $L$  with more symbols than  $D$  has state. If it is accepted some state must have been repeated. Since the DFA cannot keep a memory of its execution history, it is possible to traverse the loop over and over again to accept other (longer) words.



Example:  $a^i b^i$  is not regular.

Assume it is regular. Let  $x = a^i b^i$  with  $i = n$  (number of states). Now apply the pumping lemma. Then  $u = a^j, v = a^k, w = a^l b^i$ . Thus  $\forall m > 0 : a^j a^{mk} a^l b^i \in L$ .

So  $D$  also accepts  $a^* b^i$ . **Contradiction.**

### Regular Grammars (cont.)

If all productions in  $P$  have the form  $A \rightarrow wB$  or  $A \rightarrow w$  where  $A, B \in N$  and  $w$  is a word over  $T$  the grammar is regular and right-linear. Alternatively we can demand that all productions have the form  $A \rightarrow Bw$  or  $A \rightarrow w$ . In this case the grammar is regular and left-linear.

**A regular grammar generates a regular language.**

Example:

$G = (\{X, Y\}, \{a, b\}, \{X \rightarrow aX, X \rightarrow aY, Y \rightarrow bY, Y \rightarrow b\}, X)$  generates  $L = \{a^i b^j \mid i > 0, j > 0\}$ .

## Output in Automata

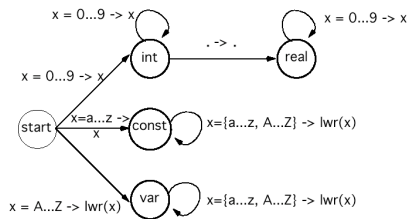
In practice our FSAs also need to generate output, i.e. they synthesize the tokens and should return a token type. There are two different ways of generating output in state automata (or state machines).

**Mealy machines** are FSAs in which each transition generates an output.

**Moore machines** are FSA in which each state generates an output.

Note: We use the state names of the different end states to derive the classification of the recognized token.

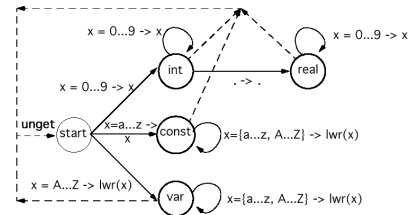
Example: A Mealy machine to recognize, classify and normalize numbers and identifiers.



## Lookahead

If we implement a DFA for the lexical level, at least one character of lookahead is needed, because we operate on a sequence of word (i.e. the character which terminates a token can be the first character of the next token and must be available to the next recognition step).

Example: input is **word1234rest**. If we recognize numbers and word consisting only of letters and blanks are not significant, this sequence consists of 3 tokens.



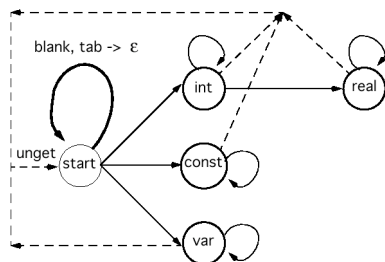
## Whitespace Removal

The lexer can remove redundant characters.

Fully redundant characters (e.g. blanks in Fortran) can even be stripped before lexing.

Partially redundant characters (e.g. token-delimiting whitespaces like blanks, tabs, etc. in most languages) have to be removed by the lexer.

As these can only occur between tokens this is trivial:



## Scanning in ML

We are now ready to implement a small scanner in ML. Our scanner will recognize integer numbers, real numbers, some operators, variables (only alpha characters, all uppercase) and constant identifiers (only alpha characters, all lowercase).

We declare a type for the different token types as well as some other additional types:

```
exception LEXER_ERROR of string;

datatype statename = ERROR | START | ID | VAR
                  | INT | REAL | OP;

type inputtype = char;

val whitespaces = [" ", "\t", "\n"];

type state = statename * inputtype * statename;

type statetable = state list;
```

## Scanning in ML (cont.)

The FSA itself is specified by its final states and transition table

```

val endstates = [VAR, ID, INT, REAL, OP];

val automaton =
  [ ( START, #" ", START),

    ( START, #"+", OP),
    ( START, #"-", OP),
    ( START, #"*", OP),
    ( START, #"/", OP),

    ( START, #"A", VAR ),
    ( START, #"B", VAR ),
    ( START, #"C", VAR ), ...

    ( VAR, #"A", VAR ),
    ( VAR, #"B", VAR ),
    ( VAR, #"C", VAR ), ...

    ( START, #"a", ID ),
    ( START, #"b", ID ),
    ( START, #"c", ID ), ...

    ( ID, #"a", ID ),
    ( ID, #"b", ID ),
    ( ID, #"c", ID ), ...

    ( START, #"1", INT ),
    ( START, #"2", INT ), ...

    ( INT, #"1", INT ),
    ( INT, #"2", INT ), ...

    ( INT, #".", REAL ),

    ( REAL, #"1", REAL ),
    ( REAL, #"2", REAL ), ...

  ];

```

## Scanning in ML (cont.)

We need a function that finds the applicable transition (if any) and another one that checks whether we are in an endstate (if there is no transition for the lookahead character):

```

fun findTransition (state, symbol) [] =
  (false, ERROR)
  | findTransition (x as (thisState, thisSymbol))
    ((state, symbol, follow)::Rest) =
    if thisState=state andalso thisSymbol=symbol
    then (true, follow)
    else findTransition x Rest;

fun checkEndstate stateName chars c =
  let val token = implode(reverse chars) in
    if (member stateName endstates)
    then (stateName, token)
    else raise LEXER_ERROR (str(c)^token)
  end;

```

## Scanning in ML (cont.)

With these the complete scanner can be implemented as:

```

fun skip_ws [] = []
  | skip_ws (all as c::cs) = if (member c whitespaces)
    then (skip_ws cs)
    else all;

fun lex1 Cs automaton =
  lex2 START nil (skip_ws Cs) automaton
and lex2 currentState soFar (C::Cs) automaton =
  (let val {1=found, 2=followState}
    = findTransition (currentState, C) automaton
  in if found then lex2 followState (C::soFar) Cs
    automaton
    else (checkEndstate currentState soFar C)
    :: (lex1 (C::Cs) automaton)
  end
  handle LEXER_ERROR t =>
    (print("Illegal token: "^t^"\n");
    lex1 Cs automaton)
)
| lex2 currentState soFar [] automaton =
  [(checkEndstate currentState soFar (chr 0))]
  handle LEXER_ERROR t =>
    (print("Unexpected end of input\n");[]);

fun lex S = lex1 (explode S) automaton;

```

Note how redundant whitespace is skipped in the start state.

## Scanner Generator

Luckily we don't have to go to this effort everytime that we need a scanner in ML. Instead we can use a scanner generator.

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:

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

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.

## ML Lex

ML Lex is a complete scanner generator in ML in which the required screening functions can be embedded in ML. It is part of the standard ML distribution.

Documentation can also be found at:

<http://www.cs.princeton.edu/~appel/modern/ml/>

ML Lex reads a lexer definition file `*.lex` and produces a standard ML file `*.sml` as output which implements this scanner.

An ML Lex specification has the format

```
user declarations
%% ML-Lex definitions
%% rules
```

The rules are the core part of the specification. Each rule has the form

```
regular expression => ( code );
```

and defines a token type (with a regular expression) and some code to be executed when this token type is recognized.

## ML Lex (cont.)

The code part can use some special values: `yypos` and `yylineno` give the current character position and line number. The function `REJECT()` can be called to let the current rule application be rejected (as if it had not matched).

`lex()` is another useful function. It invokes the lexer recursively and can be used to skip whitespace with a rule like:

```
[\ \t\n]+ => ( lex());
```

## Regular expressions in ML Lex

any character except for the special characters

```
? * + | ( ) ^ $ / ; . = < > [ { " \.
```

`.` stands for any newline character

`\b` backspace

`\n` newline

`\t` tab

`\ddd` where `ddd` is a 3 digit decimal escape

`[abcde...]` denotes exactly one of `a, b, c, d, e, ...`

sequences of characters may be enclosed in double quotes

`{name}` refers to a defined named regular expression (see below)

`(r)`, `r*`, `r+`, `r1|r2` are the usual regular expression derived from other regular expressions `r, r1, r2`.

`r?` denotes zero or one occurrences

`r{n}` denotes exactly `n` occurrences of `r`

## Declarations in ML Lex

The *user declarations* contain user-defined code and must specify two values: a type `lexresult` (the token types) and a function to be called when the "end of file" is reached.

The *ML-Lex definitions section* can contain various declarations for the lexer generator (see documentation).

In this section named regular expressions can also be defined via

```
identifier = regular expression
```

The following commands are also available:

`%reject` to create the `REJECT()` function

and

`%count` to create the function to count newlines using `yylineno`

using either of these slows down the lexer significantly!

### An ML Lex example

The following is an ML-Lex definition of a scanner for arithmetic expressions (adapted from the ML-Lex documentation):

```
datatype lexresult= DIV | EOF | EOS | ID of string |
                  LPAREN | NUM of int | PLUS | PRINT |
                  RPAREN | SUB | TIMES

val linenum = ref 1
val error = fn x => print(x ^ "\n")
val eof = fn () => EOF
%%
%structure CalcLex
alpha=[A-Za-z];
digit=[0-9];
ws = [\ \t];
%%
\n      => (linenum := 1 + !linenum; lex());
{ws}+   => (lex());
"/"     => (DIV);
";"     => (EOS);
"("     => (LPAREN);
{digit}+ => (NUM (foldl
              (fn(a,r)=>ord(a)-ord("#0")+10*r) 0
              (explode yytext)));
")"     => (RPAREN);
"+"     => (PLUS);
{alpha}+ => (if yytext="print" then PRINT else ID yytext);
"-"     => (SUB);
"*"     => (TIMES);
```

### Using the Generated Scanner

You can now simply load the generated lexer (with extension \*.lex.sml)

```
Standard ML of New Jersey ...
val it = () : unit
- use "calcLex.lex.sml";
[opening calcLex.lex.sml]
GC #0.0.0.0.1.18: (0 ms)
GC #0.0.0.0.1.2.28: (10 ms)
structure CalcLex :
  sig
    structure Internal : <sig>
      structure UserDeclarations : <sig>
        exception LexError
        val makeLexer : (int -> string) ->
          unit -> Internal.result
      end
    end
  end
val it = () : unit
```

To perform the scanning, open a file and generate a lexer. The constructor function is passed a function argument

```
- val inputfile = TextIO.openIn("testfile");
val it = - : TextIO.instream
- val lexer = CalcLex.makeLexer(
  fn n => TextIO.inputLine inputfile);
val lexer = fn : unit -> CalcLex.Internal.result
```

### ML Lex example

To use ML-Lex you must first generate the scanner from its definition:

```
bruce_27%ml-lex calcLex.lex
```

```
Number of states = 18
Number of distinct rows = 5
Approx. memory size of trans. table = 645 bytes
```

This generates a file `calcLex.lex.sml` which contains the executable scanner.

### Using the Generated Scanner

We assume the input testfile has contents

```
1 + ( 3 * 4 ) / abc + 2
```

Each successive call of `lexer()` will yield one token

```
- lexer();
val it = NUM 1 : CalcLex.UserDeclarations.lexresult
- lexer();
val it = PLUS : CalcLex.UserDeclarations.lexresult
- lexer();
val it = LPAREN : CalcLex.UserDeclarations.lexresult
- lexer();
val it = NUM 3 : CalcLex.UserDeclarations.lexresult
- lexer();
val it = TIMES : CalcLex.UserDeclarations.lexresult
- lexer();
val it = NUM 4 : CalcLex.UserDeclarations.lexresult
- lexer();
val it = RPAREN : CalcLex.UserDeclarations.lexresult
- lexer();
val it = DIV : CalcLex.UserDeclarations.lexresult
- lexer();
val it = ID "abc" : CalcLex.UserDeclarations.lexresult
- lexer();
val it = PLUS : CalcLex.UserDeclarations.lexresult
- lexer();
val it = NUM 2 : CalcLex.UserDeclarations.lexresult
- lexer();
val it = EOF : CalcLex.UserDeclarations.lexresult
```

## Flex

**flex** is a similar public domain scanner generator for UNIX systems. It works with C instead of ML.

**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` which contains the function `yylex` which can be called by other programs such as a parser.

A flex specification has almost the same form as an ML Lex specification (ML Lex was modelled on Flex), but Flex is (still) a bit more powerful and has more flexible expressions.

## Summary

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.
- Implementing a lexer in ML.
- Scanner generators: MLLex and 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.
- Read the MLLex documentation at <http://www.cs.princeton.edu/~appel/modern/ml/>.
- Implement a lexer with MLLex that can scan the rules of an MLLex definition file.
- extend the scanner that was given in plain ML in the lectures to call arbitrary ML functions when a token has been recognized. These functions should take the identifier type (statename) as a single argument and return a boolean (successful execution). The functions to be called should be specified in a value of type  $(state\ name \ * \ (state\ name \ \rightarrow \ bool)) \ list$ .

## 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.

## Reminder: Syntax Analysis

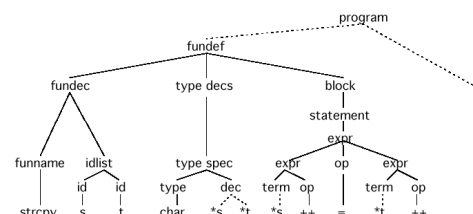
The task of the syntax analysis is to recover the hierarchical (recursive) structure from the flat token sequence.

### Example

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

```
[ res(strcpy), (, 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.



## Syntactical Structures

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.

Obviously, a regular grammar is not powerful enough to express this recursive structure of statements. A derivation according to regular grammar always has a linear structure (this is obvious from the form of its productions). Thus a more flexible form of grammar is needed.

## 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 **productions** each of form  $X \rightarrow \alpha$  where  $X \in N$  and  $\alpha$  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:

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

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  $\alpha, \beta, \gamma$  denote **sentences**, that is sequences from  $(N \cup T)^*$ .

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

$\alpha \Rightarrow^* \beta$  indicates that  $\beta$  can be derived from  $\alpha$  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.

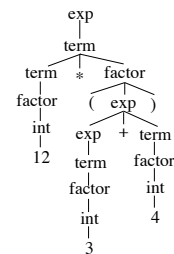
A **derivation** can be represented using a **parse** or **syntax** tree:

- 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

## Parse Trees

$$\{\alpha \mid \alpha \in T^* \text{ and } S \Rightarrow^* \alpha\}.$$

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



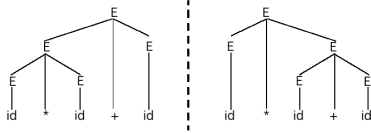
Notice that the parse tree can abstract away from the order in which production rules are used.

## Ambiguity

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

For example, the following grammar is ambiguous:

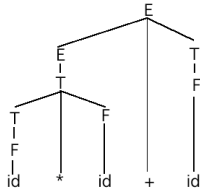
$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$



However, the modification

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

allows only a single parse tree.



## Disambiguation

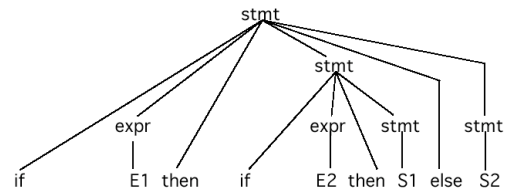
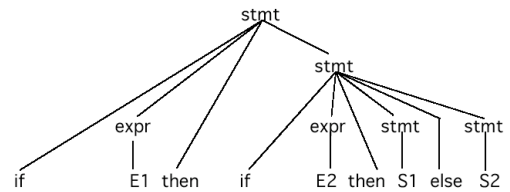
Some context-free languages are **inherently ambiguous** while in other cases it can be eliminated.

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

is ambiguous since

$$\text{if } e1 \text{ then if } e2 \text{ then } s1 \text{ else } s2$$

has two parse trees.



## Disambiguation (cont.)

Disambiguation of course 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

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

## Limitations of Context-free Grammars

Many programming languages constructs cannot be expressed or defined by context-free grammars. Consider the following examples:

- All identifiers have to be declared before they are used.  
Corresponding formal language:  $L = \{wcv \mid w \in (a \mid b)^*\}$
- Formal parameters in procedure declarations must agree with their usage in procedure calls.  
Corresponding formal language:  $L = \{a^n b^m c^n d^m \mid n, m \geq 1\}$

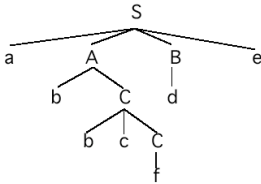
Such checks therefore have to be left to the semantic analysis phase.

Using the (more complex) *Pumping Lemma* for context-free languages, it can be proven that the above languages are not context-free.

Beware:  $L = \{a^n b^m c^m d^n \mid n, m \geq 1\}$  is context-free! Why?

## Top-Down Parsing

In **top-down parsing** the parse tree is built from the root downwards interpreting productions from left to right. Example

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow bC \\ B &\rightarrow d \\ C &\rightarrow bcC \mid f \end{aligned}$$


leftmost:

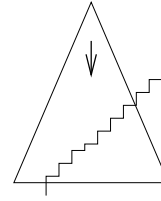
$$S \rightarrow aABe \rightarrow abCBe \rightarrow abbcCBe \rightarrow abbcfBe \rightarrow abbcfde.$$

rightmost:

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

leftmost (rightmost) derivations always replace the leftmost (rightmost) non-terminal in the current sentential form.

## Predictive Top-down Parsing



top-down parsing

The parser scans the input-left-to-right one token at a time.

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.

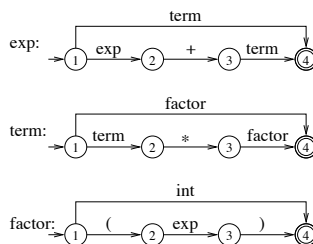
## Transition Diagrams

In practice computational languages are often described by **transition diagrams** instead of grammars. However, this is just a convenient graphical representation of the grammar.

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.
- Finally, if  $x$  is  $\epsilon$  it simply moves to state  $t$  without reading any input.

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

## Elimination of Left Recursion

The preceding transition diagram is 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  $\alpha$ .

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

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

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

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

## Elimination of Cycles and $\epsilon$ -Productions

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

A context-free grammar can be made  $\epsilon$ -free (i.e. it does not contain any  $\epsilon$ -productions) if the language generated by  $G$  does not contain  $\epsilon$ .

- determine all non-terminals  $X$  for which  $X \Rightarrow^* \epsilon$ . We call these non-terminals *nullable*.
- replace each production  $p$  of the form  $A \rightarrow B_1 B_2 \dots B_n$  by a set of productions that is composed of copies of  $p$  with each possible combination of nullable non-terminals removed on the LHS.

A grammar is not cycle-free if  $A \Rightarrow^+ A$  for some non-terminal  $A$ . Obviously, a cycle in a derivation performs no useful function. How can cycles be removed automatically?

## 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 replaced 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**

## 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

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

if we encounter an **if** which production do we use?

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

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

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

The above grammar will be rewritten to:

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

## Recursive-Descent Parsing (Example Grammar)

We now implement a recursive descent parser for our transformed expression grammar:

```
exp   → term exp1
exp1  → + exp
exp1  → - exp
exp1  → ε
term  → factor term1
term1 → * term
term1 → / term
term1 → ε
factor → id
factor → (Exp)
```

## Recursive-Descent Parsing in ML

For the scanner stage of the following recursive descent parser we use the scanner that we have generated in the last lecture using ML-Lex.

First, we make the internal structure of the lexer (in particular the tokens) available and declare a type for the syntax tree. ML-Lex uses a datatype `CalcLex.Internal.result` for the token types. We also need to access `CalcLex.UserDeclarations` which contains the individual constructors for the tokens.

```
open CalcLex.Internal;
open CalcLex.UserDeclarations;

datatype opNode = OP of result;

datatype synTree =
  Empty
  | EXP of synTree * synTree
  | EXP1 of opNode * synTree
  | TERM of synTree * synTree
  | TERM1 of opNode * synTree
  | FACTOR of result;
```

## Recursive-Descent Parsing in ML (cont.)

Each production is implemented by a single function that passes back a tuple consisting of the syntaxtree for this production application and the unprocessed rest of the input.

```
fun exp tokenList =
  let val (termTree, rest1) = term tokenList in
    let val (exp1Tree, rest) = exp1 rest1 in
      (EXP(termTree, exp1Tree), rest)
    end
  end
and exp1 (PLUS::more) =
  let val (expTree, rest) = exp more in
    (EXP1(OP(PLUS), expTree), rest)
  end
| exp1 (SUB::more) =
  let val (expTree, rest) = exp more in
    (EXP1(OP(SUB), expTree), rest)
  end
| exp1 rest = (Empty, rest)
```

continued on the next page...

## Recursive-Descent Parsing in ML (cont.)

continued from previous page...

```
and term tokenlist =
  let val (factorTree, rest1) = factor tokenlist in
    let val (term1Tree, rest) = term1 rest1 in
      (TERM(factorTree, term1Tree), rest)
    end
  end
and term1 (DIV::more) =
  let val (termTree, rest) = term more in
    (TERM1(OP(DIV), termTree), rest)
  end
| term1 (TIMES::more) =
  let val (termTree, rest) = term more in
    (TERM1(OP(TIMES), termTree), rest)
  end
| term1 rest = (Empty, rest)
and factor (ID(s)::rest) = (FACTOR(ID(s)), rest)
| factor (NUM(n)::rest) = (FACTOR(NUM(n)), rest)
| factor (LPAREN::more) =
  let val (expTree, (RPAREN::rest)) = exp more in
    (expTree, rest)
  end;
```

Note how (in the general case) the functions have to be chained by `and` because the productions could be mutually recursive.

## Recursive-Descent Parsing in ML (cont.)

Finally, we need some help functions to couple the scanner and the recursive descent parser:

```
exception SYNTAX_ERROR of lexresult list;

fun allTokens (lexer) =
  let val token = (lexer():lexresult) in
    if token=EOF
    then [EOF]
    else token::allTokens(lexer)
  end;

fun parse () = parse1 []
and parse1 [] =
  let val infile = (TextIO.openIn("testfile")) in
    let val lexer =
        CalcLex.makeLexer(
          fn n => TextIO.inputLine infile ) in
      let val (parseTree, rest) = exp (allTokens(lexer)) in
        if rest=[ EOF ] then parseTree
        else raise SYNTAX_ERROR(rest)
      end
    end
  end;
end;
```

## Programming Language Implementation IV—V

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.

## 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 include function applications (like  $\sin(x)$ )
- Build a recursive descent parser for the 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

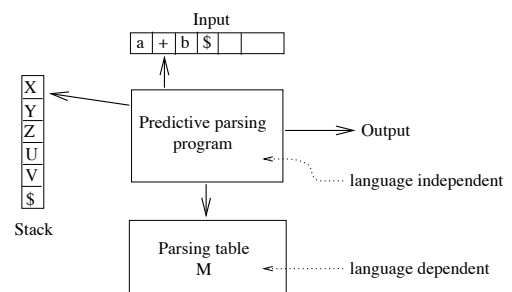
$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd$$

## 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.

### Predictive Parsing Algorithm

Independently of the target language, a predictive parser is controlled by a simple program which does the following:

```

initialize input to  $w$ ;
initialize stack to  $S$ ;
repeat{
   $X := \text{top}()$ ;
   $a := \text{currentSymbol}()$ ;
  if  $X = \$$  or  $\text{isTerminal}(X)$  then
    if  $X = a$  then {  $\text{pop}()$ ;  $\text{advanceInput}()$ ; }
    else raise syntaxError;
  else
    if  $M[X, a] = X \rightarrow Y_1, \dots, Y_k$  then {
       $\text{pop}()$ ;
       $\text{push}(Y_k); \dots, \text{push}(Y_1)$ ;
    }
    else raise syntaxError; (*  $M[X, a]$  empty *)
}
until  $X = \$$ 

```

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 determines the production to be applied from the parsing table.

### Example (Cont.)

Consider parsing the symbol string

**int + int**

Stack	Input	Production
exp \$	int + int \$	P1
term exp' \$	int + int \$	P4
factor term' exp' \$	int + int \$	P7
int term' exp' \$	int + int \$	advance()
term' exp' \$	+ int \$	P6
exp' \$	+ int \$	P2
+ exp \$	+ int \$	advance()
exp \$	int \$	P1
term exp' \$	int \$	P4
factor term' exp' \$	int \$	P7
int term' exp' \$	int \$	advance()
term' exp' \$	\$	P6
exp' \$	\$	P3
\$	\$	accept!

### Example

Recall the grammar

```

exp   → term exp' (P1)
exp'  → +exp      (P2)
exp'  → ε         (P3)
term  → factor term' (P4)
term' → *term     (P5)
term' → ε         (P6)
factor → int      (P7)
factor → (exp)   (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	

### Table-driven Predictive Parsing in ML

First we have to declare data types for the terminal and non-terminal symbols of the grammar and create data structures to store the parse table and the productions.

Note that only the right-hand side of the productions need to be stored (as the left-hand side is accessed via the parse table).

```

datatype lexResult = PLUS | TIMES | INT |
                  LPAREN | RPAREN | EOF ;
datatype nonTerminal = EXP | EXP1 | TERM | TERM1 |
                    FACTOR ;

datatype token = NT of nonTerminal | T of lexResult;

val productions =
  [[NT(TERM), NT(EXP1)],
   [T(PLUS), NT(EXP)],
   [],
   [NT(FACTOR), NT(TERM1)],
   [T(TIMES), NT(TERM)],
   [],
   [T(INT)],
   [T(LPAREN), NT(EXP), T(RPAREN)]]];

```

### Table-driven Predictive Parsing in ML (cont.)

Note how the parse table and its matching function can be implemented in an integrated form as a function using pattern matching. The parseTable lookup returns an index into the list of productions.

```
fun parseTable EXP      INT      = 1
  | parseTable EXP      LPAREN   = 1
  | parseTable EXP1     PLUS     = 2
  | parseTable EXP1     RPAREN   = 3
  | parseTable EXP1     EOF      = 3
  | parseTable TERM     INT      = 4
  | parseTable TERM     LPAREN   = 4
  | parseTable TERM1    PLUS     = 6
  | parseTable TERM1    TIMES    = 5
  | parseTable TERM1    RPAREN   = 6
  | parseTable TERM1    EOF      = 6
  | parseTable FACTOR   INT      = 7
  | parseTable FACTOR   LPAREN   = 8
  | parseTable _        _        = raise SYNTAX_ERROR;
```

```
fun pushAll [] stack = stack
  | pushAll (x::xs) stack = pushAll xs (x::stack);

fun parse () =
  let val allTokens = [INT, PLUS, INT, EOF] in
    let val rest = (step [NT(EXP)] allTokens) in
      if rest=[EOF] then print "accept\n"
      else raise SYNTAX_ERROR
    end
  end
end;
```

### Table-driven Predictive Parsing in ML (cont.)

The function `step` implements the core function of the table parser: It compares the top of the stack (first argument) with the current symbol of the input (second argument), pops the stack and either

- advances the input or
- performs a table lookup and pushes the right-hand side of the applicable production on the stack or
- raises a syntax error

```
fun step ([]:token list) rest = rest
  | step _ [] = raise SYNTAX_ERROR
  | step (T(top)::RestStack) (this::moreInput) =
    if (top=this) then
      (print "advance()\n";
       step RestStack moreInput
      )
    else raise SYNTAX_ERROR
  | step (NT(top)::RestStack) (this::moreInput) =
    let val index = parseTable top this in
      (print ("executing production no. " ^
              Int.toString(index) ^ "\n");
       step (pushAll (List.rev
                     (List.nth(productions, index-1)))
            RestStack)
            (this::moreInput)
      )
    end;
```

### How to Construct a Predictive Parser

To construct a predictive table parser we need to compute two sets:  $FIRST(X)$  and  $FOLLOW(X)$  for every grammar symbol  $X$ .

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

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

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

```
for each production  $A \rightarrow w$ 
  for each terminal  $a \in FIRST(w)$  add  $A \rightarrow w$  to  $M[A, a]$ ;
  if  $\epsilon \in FIRST(w)$  then
    for each terminal  $b \in FOLLOW(A)$  add  $A \rightarrow w$  to  $M[A, b]$ ;
  if  $\epsilon \in FIRST(w) \wedge \$ \in FOLLOW(A)$  then
    add  $A \rightarrow w$  to  $M[A, \$]$ ;
set all other entries to empty (error);
```

### Computing FIRST

The table construction makes use of two functions **FIRST** and **FOLLOW** associated with any grammar.

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

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  $\epsilon$  to  $FIRST(X)$ .
- If  $X \rightarrow Y_1Y_2 \dots Y_k$  is a production, then  $FIRST(X) := FIRST(Y_1Y_2 \dots Y_k)$

For a string  $\alpha = X_1, \dots, X_n$ , we can compute  $FIRST(\alpha)$  as follows:

1.  $FIRST(\alpha) = FIRST(X_1) - \{\epsilon\}$
2. for  $i = 2 \dots n$  : if  $\forall_{j < i} \epsilon \in FIRST(X_j)$  then add  $FIRST(X_i) - \{\epsilon\}$  to  $FIRST(\alpha)$ .
3. add  $\epsilon$  to  $FIRST(\alpha)$  if  $\forall_i : \epsilon \in FIRST(X_i)$ .

### Computing 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 rightmost symbol in some sentence, then  $\$ \in FOLLOW(A)$ .

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

**Exercise:** Give the  $FOLLOW$  sets for the previous grammar.

For example, consider the grammar:

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

### Construction Example

Recall

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

We have that

- $FIRST(exp) = FIRST(term) = FIRST(factor) = \{(\mathbf{int})\}$
- $FIRST(exp') = \{+, \epsilon\}$
- $FIRST(term') = \{*, \epsilon\}$
- $FOLLOW(exp) = FOLLOW(exp') = \{), \$\}$
- $FOLLOW(term) = FOLLOW(term') = \{+, ), \$\}$
- $FOLLOW(factor) = \{+, *, ), \$\}$ .

The parsing table for this grammar is

	<b>int</b>	+	*	(	)	\$
<i>exp</i>						
<i>exp'</i>						
<i>term</i>						
<i>term'</i>						
<i>factor</i>						

## 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:

- pop();
- push( $E$ ); push(+); 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.*

## Error Correction

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

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.

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

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

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

## 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 parse table generated for a lookahead of one symbol has no multiply-defined 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  $\alpha$  and  $\beta$  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.

More generally we talk about  $LL(k)$  grammars. These can be predictively parsed with  $k$  symbol lookahead.

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

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

The parsing table with error correction for this grammar is

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

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.

### Summary

We have looked at:

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

### Homework

- Read Section 4.4 of Aho et al.
- Modify the grammar for arithmetic expressions to include division and subtraction and identifier.
- Extend the table-driven predictive parser given in the lecture for this extended grammar and couple it to the `calcLex` lexer.
- Extend the table-driven predictive parser given in the lecture with error correction.

### Error Correction – Example (Cont.)

Consider parsing the symbol string

)**int** + \***int**

## Programming Language Implementation VI

In this lecture we will look at **semantic analysis** and in particular at

- **Attribute grammars**
- **Syntax-directed Translation**

The material is (loosely) based on Aho et al Chapter 5.

## Semantic Analysis

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

- typically determines the **kind** of each identifier
- performs **type checking** and **type inference**, and
- adds this information to the symbol table.

It also checks that

- variables are declared before use,
- variables are declared only once (within a particular scope),
- type compatibility and required coercions,
- matching of actual with formal parameters,
- resolves overloaded operator symbols
- ...

A separate phase for semantic analysis is required because context-free grammars are not powerful enough to check these properties which they are inherently **context sensitive**. *Remember:*  $L = \{wcv \mid w \in (a \mid b)^*\}$ , for example, is not a context-free language.

Semantic analysis is often done in an ad hoc manner, but **attribute grammars** can be used to formalize it.

## Attribute Grammars

**Attribute grammars** are due to Knuth in 1968.

They extend the normal grammar mechanism by adding attributes to non-terminals. These attributes can mainly be used for two purposes:

- to compute structures (e.g. syntax-trees) to be returned by the grammar and
- to steer the application of productions by providing additional information when calling a production.

There are two attribute types: *inherited* attributes and *synthesized* attributes. Synthesized attributes are composed by a production. Inherited attributes are provided when a production is called and tested or used to compute synthesized attributes.

## Attribute Grammars (cont.)

With each grammar production  $A \rightarrow \alpha$  we associate a set of semantic rules of the form

$$b := f(c_1, \dots, c_n)$$

where  $f$  is a (usually side effect-free) function,  $c_1, \dots, c_n$  are attributes of the symbols in the rule and either:

- $b$  is an attribute of  $A$ , in which case  $b$  is a **synthesized** attribute.
- $b$  is an attribute of one of the symbols in  $\alpha$ , in which case  $b$  is an **inherited** attribute.

More generally, a production could also specify

- **conditions** on attribute values for a production to be applicable,
- **procedures** to be evaluated which, for example, might update the symbol table.

A parse tree showing the values of the attributes at each node is said to be **annotated** or **decorated**.

## Computation of Attributes

- Inherited attributes in a production  $P : X \rightarrow X_1, \dots, X_n$  for a non-terminal  $X_i$  are computed from
  1. inherited attributes of  $X$ ,
  2. synthesized attributes on the right-hand side in  $X_1, \dots, X_{i-1}$ ,
  3. attributes of terminals on the right-hand side in  $X_1, \dots, X_{i-1}$ .
- Synthesized attributes in a production  $P : X \rightarrow X_1, \dots, X_n$  for a non-terminal  $X$  are computed from
  1. inherited attributes of  $X$ ,
  2. synthesized attributes on the right-hand side of  $P$ ,
  3. attributes of terminals on the right-hand side of  $P$ .

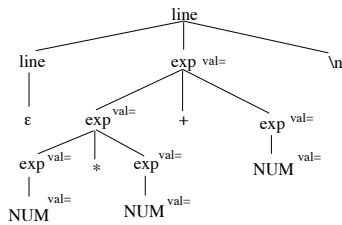
### Attribute Grammar – Synthesized

A simple example of an attribute-grammar that uses only synthesized attributes is:

Production	Semantic Rules
$line \rightarrow \epsilon$	
$line \rightarrow line \ exp \ \backslash n$	$print(exp.val)$
$exp_0 \rightarrow exp_1 + exp_2$	$exp_0.val := exp_1.val + exp_2.val$
$exp_0 \rightarrow exp_1 - exp_2$	$exp_0.val := exp_1.val - exp_2.val$
$exp_0 \rightarrow exp_1 * exp_2$	$exp_0.val := exp_1.val * exp_2.val$
$exp_0 \rightarrow exp_1 / exp_2$	$exp_0.val := exp_1.val / exp_2.val$
$exp_0 \rightarrow (exp_1)$	$exp_0.val := exp_1.val$
$exp_0 \rightarrow NUM$	$exp_0.val := NUM.val$

In this grammar  $exp$  and  $NUM$  have a single synthesized attribute  $val$  and  $line$  and the remaining terminal symbols have no attributes.

The annotated parse tree for  $3 * 5 + 4 \backslash n$  is:



### Expressive Power of Attribute Grammars

Earlier we have pointed out that attribute grammars have higher expressive power than normal context-free grammars.

Example:

We know that  $L = a^n b^m c^n d^m$  is not a context-free language.

It is, of course, easy to write an attribute grammar for  $L$ :

Production	Semantic Rules
$S \rightarrow A B C D$	if $C.n = A.n \wedge D.n := B.n$
$A \rightarrow \epsilon$	$A.n := 0$
$A_1 \rightarrow a A$	$A_1.n := A_2.n + 1;$
$B \rightarrow \epsilon$	$B.n := 0$
$B_1 \rightarrow b B_2$	$B_1.n := B_2.n + 1;$
$C \rightarrow \epsilon$	$C.n := 0$
$C_1 \rightarrow c C_2$	$C_1.n := C_2.n + 1;$
$D \rightarrow \epsilon$	$D.n := 0$
$D_1 \rightarrow d D_2$	$D_1.n := D_2.n + 1;$

### Attribute Grammar – Inherited

A simple example of an attribute-grammar that uses only inherited attributes is:

Production	Semantic Rules
$exp_1 \rightarrow term + exp_2$	$term.rep := exp_1.rep;$ $exp_2.rep := exp_1.rep;$
$exp \rightarrow term$	$term.rep := exp.rep$
$term \rightarrow NUM$	$NUM.rep := term.rep$
$NUM \rightarrow ZERO \mid ONE \mid \dots \mid NINE$	if $NUM.rep = 'words'$
$NUM \rightarrow 0 \mid 1 \mid \dots \mid 9$	if $NUM.rep = 'digits'$

In this grammar  $term$ ,  $exp$  and  $NUM$  have a single inherited attribute  $rep$  that switches between two types of representations in the terminals.

### Attribute Grammar – Mixed Attributes

In particular grammars that have been left-factored or made LL(k) often lose the “intuitive” structure of the grammars, so that even simple computations require a mix of inherited and synthesized attributes.

Consider our grammar for arithmetic expressions.

$exp \rightarrow$	$term \ exp'$ $exp'.v1 := term.v; exp.v := exp'.v;$
$exp' \rightarrow$	$+exp$ $exp'.v := exp'.v1 + exp.v;$
$exp' \rightarrow$	$\epsilon$ $exp'.v := exp'.v1;$
$term \rightarrow$	$factor \ term'$ $term'.v1 := factor.v; term.v := term'.v;$
$term' \rightarrow$	$*term$ $term'.v := term'.v1 * term.v;$
$term' \rightarrow$	$\epsilon$ $term'.v := term'.v1;$
$factor \rightarrow$	<b>int</b> $factor.v := int.v;$
$factor \rightarrow$	$(exp)$ $factor.v := exp.v;$

## Attribute Dependency Graph

A sentence such as  $x := y+z$  has a **dependency graph** detailing how each attribute depends on the other attributes.

An attribute grammar is **well-defined** if every attribute is defined and for no sentence does the dependency graph contain a cycle. (ie it must be a **dag**-directed acyclic graph).

For every dag, we can list the attributes so that the attribute comes after those attributes on which it depends. (Using **topological sorting**).

However we would like a general (recursive) way of computing attributes while traversing the parse tree.

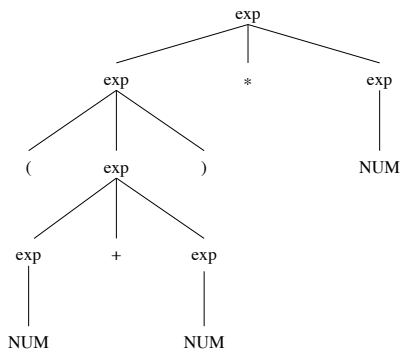
## Construction of Structure Trees

Earlier we noted that usually the parser does not build the full parse tree but rather strips it to the essential **structure tree**, (sometimes called an **abstract syntax tree**). This process can be specified using attributes.

Production	Semantic Rules
$exp_0 \rightarrow exp_1 + exp_2$	$exp_0.tree := tree('+', exp_1.tree, exp_2.tree)$
$exp_0 \rightarrow exp_1 - exp_2$	$exp_0.tree := tree('-', exp_1.tree, exp_2.tree)$
$exp_0 \rightarrow exp_1 * exp_2$	$exp_0.tree := tree('*', exp_1.tree, exp_2.tree)$
$exp_0 \rightarrow exp_1 / exp_2$	$exp_0.tree := tree('/', exp_1.tree, exp_2.tree)$
$exp_0 \rightarrow (exp_1)$	$exp_0.tree := exp_1.tree$
$exp_0 \rightarrow NUM$	$exp_0.tree := leaf(NUM.val)$

The function *leaf* is the type constructor for a leaf node and *tree* is the constructor for non-leaf nodes.

For example, evaluation of the parse tree  $(2 + 4) * 3$  leads to the abstract syntax tree:



## Attribute Grammar Generators

There are many tools available for automatically generating a semantic analyzer from an attribute grammar. These include

- **Elegant**. The elegant system. Philips Research.
- **Eli**. The eli system, compiler construction made easy. University of Colorado at Boulder, University of Paderborn, Macquarie University in Sydney.
- **FNC-2**. The fnc-2 attribute grammars system. Didier Parigot, Oscar project, INRIA Rocquencourt.
- **FUN**. The fun transformation system. Attribute Grammar Based Transformation Systems.

Search the Web if you are interested. A good starting point is:  
<http://www-rocq.inria.fr/oscar/www/fnc2/attribute-grammar-people.html>

## Syntax-directed Translation

If the mapping of the source language to the target language is comparatively simple we can employ *syntax-directed translation*.

Syntax-directed translation consist of two phases:

1. build the syntax-tree or structure tree during parsing using an attribute grammar,
2. traverse the syntax-tree recursively generating the target code.

*Example:* Translating arithmetic infix expressions to RPN code

First we need to declare a datatype for the instruction list.

```
datatype instruction =  
  PUSH of result | TIMES_OP | DIV_OP | PLUS_OP | SUB_OP ;
```

The syntax-directed translation performs a suffix traversal of the parse tree. Note how the left-factored structure of the grammar forces us to process the code in a rather unnatural fashion: every production generates two fragments of code - the **first** argument which essentially represents the fragment completing the last arithmetic expression/term and the **rest** argument representing the remainder of the expression or term.

## Syntax-directed Translation (cont.)

```
fun genCode Empty = ([], [])  
  | genCode (EXP(termTree, exp1Tree)) =  
    let val (tFirst, tRest) = genCode termTree in  
      let val (eFirst, eRest) = genCode exp1Tree in  
        ((tFirst @ tRest), (eFirst @ eRest))  
      end  
    end  
  | genCode (EXP1(OP(PLUS), expTree)) =  
    let val (first, Rest) = genCode expTree in  
      ((first @ [PLUS_OP]), Rest)  
    end  
  | genCode (EXP1(OP(SUB), expTree)) =  
    let val (first, Rest) = genCode expTree in  
      ((first @ [SUB_OP]), Rest)  
    end  
  | genCode (TERM(factorTree, term1Tree)) =  
    (* note that the factor could in fact be  
     * an expression in brackets. In this case  
     * we need to collect first, too  
     *)  
    let val (fFirst, fRest) = genCode factorTree in  
      let val (tFirst, tRest) = genCode term1Tree in  
        ((fFirst @ fRest), (tFirst @ tRest))  
      end  
    end  
  end
```

...continued on next page

## Syntax-directed Translation (cont.)

...continued from previous page

```
  | genCode (TERM1(OP(DIV), termTree)) =  
    let val (first, rest) = genCode termTree in  
      ((first @ [DIV_OP]), rest)  
    end  
  | genCode (TERM1(OP(TIMES), termTree)) =  
    let val (first, rest) = genCode termTree in  
      ((first @ [TIMES_OP]), rest)  
    end  
  | genCode (FACTOR(X)) = ([], [PUSH(X)]);  
  
fun translate () =  
  let val (expFirst, expRest) = genCode(parse()) in  
    (expFirst @ expRest)  
  end
```

For more complex languages, syntax-directed translation generates only intermediate code which is then further processed.

## Translation directly in the Grammar

In very simple cases there is no need to split the translation into two separate phases and syntax-directed translation can even be further simplified by unfolding the tree traversal into the attribute grammar.

Keep in mind that even for very simple languages this renders the parser very sensitive to changes in the target language.

*Example:* Translating arithmetic infix expressions to RPN code in a single phase

```
s → exp  
s.code := append(exp.first, exp.rest);  
exp → term exp'  
exp.first := append(term.first, term.rest);  
exp.rest := append(exp'.first, exp'.rest);  
exp' → +exp  
exp'.first := append(exp.first, [plus]);  
exp'.rest := exp.rest;  
exp' → ε  
exp'.first := [];  
exp'.rest := [];  
term → factor term'  
term.first := factor.code;  
term.rest := append(term'.first, term'.rest);  
term' → *term  
term'.first := append(term.first, [times]);  
term'.rest := term.rest;  
term' → ε  
term'.first := [];  
term'.rest := [];  
factor → int  
factor.code := [push(int.val)]  
factor → (exp)  
factor.code := append(exp.first, exp.code);
```

Summary

We have looked at semantic analysis, attribute grammars, abstract interpretation and syntax-directed translation.

Homework

- Read Chapter 5 of Aho et al.
- Extend the assignment grammar example and give attribute rules and conditions for the productions
  - $exp \rightarrow exp - exp,$
  - $exp \rightarrow exp \text{ div } exp,$
  - $exp \rightarrow floor(exp),$
 where *div* is integer division and *floor* takes a real and returns an integer.

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

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

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

Bottom-Up Parsing

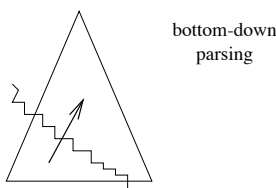
In **bottom-up parsing** the parse tree is built from the leaves upwards interpreting productions from right to left (ie. the input is reduced to the start symbol of G).

Example

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

inverted rightmost derivation:

$$abcfde \rightarrow abbcCde \rightarrow abCde \rightarrow aAde \rightarrow aABe \rightarrow S$$



In the next two lectures we will look at **table-driven** bottom-up parsers and a generic **dynamic programming** approach.

Why Bottom-Up Parsing

- Left Recursion
  - Top-down parsers loop infinitely for left-recursive grammars.
  - Bottom-up parser can process left-recursion.
- 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.
  - LR parsing, the most general known non-backtracking shift-reduce method, works for almost all the known programming language constructs (exception e.g. Haskell).
- 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.

## 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**  $h$  of a sentential form  $s$  is a substring of  $s$  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  $s$ .

$$S \xleftarrow{*}_{rrm} aAw \xleftarrow{rrm} ahw = s$$

Obviously not every RHS of a production that occurs in  $s$  is a handle.

## Handles

Consider the grammar

$$exp \rightarrow exp + exp \mid exp * exp \mid \mathbf{int} \mid (exp)$$

This has the **rightmost** derivation

$$\begin{aligned} exp &\xrightarrow{rm} \underline{exp} + exp \\ &\xrightarrow{rm} exp + \underline{exp * exp} \\ &\xrightarrow{rm} exp + exp * \underline{int_3} \\ &\xrightarrow{rm} exp + \underline{int_2} * int_3 \\ &\xrightarrow{rm} \underline{int_1} + int_2 * int_3 \end{aligned}$$

Shift-reduce parsing works as follows:

Right-sentential form	Production
$\underline{int_1} + int_2 * int_3$	$exp \rightarrow \mathbf{int}$
$exp + \underline{int_2} * int_3$	$exp \rightarrow \mathbf{int}$
$exp + exp * \underline{int_3}$	$exp \rightarrow \mathbf{int}$
$exp + \underline{exp * exp}$	$exp \rightarrow exp * exp$
$\underline{exp} + exp$	$exp \rightarrow exp + exp$
$exp$	

where the underlined symbols are **handles**.

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

## Stack Implementation 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
\$	$a_1 a_2 \dots a_n \$$

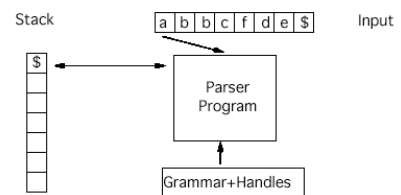
and wants to reach

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.

## Shift Reduce Algorithm



```
repeat
  repeat
    shift current input symbol on stack;
    advance input pointer
  until a handle is on top of the stack or input is empty
  reduce handle (if present) to its corresponding LHS
  (= pop handle; push LHS)
until stack==$ or input=$
if stack==$ and input=$ accept.
```

A stack is an appropriate data structure, because in SR parsing a handle will always appear on top of the stack, never inside.

### 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.

### Conflicts in Shift-Reduce Parsing

When backtracking is not used, the parser has to decide deterministically at each step which action to apply.

Some grammars produce conflicts that render the parser unable to decide deterministically. These cannot be used for deterministic shift-reduce parsing (without backtracking).

- shift/reduce conflicts: It cannot be decided whether to shift or to apply a production.
- reduce/reduce conflicts: It cannot be decided which of several productions to apply.

Example

$$\begin{aligned}
 stmt &\rightarrow \mathbf{if} \text{ expr } \mathbf{then} \text{ stmt} \\
 stmt &\rightarrow \mathbf{if} \text{ expr } \mathbf{then} \text{ stmt } \mathbf{else} \text{ stmt} \\
 stmt &\rightarrow \text{other forms} \dots
 \end{aligned}$$

When if expr then stmt is on top of the stack and else is the current input symbol, a shift/reduce conflict occurs.

### Operator Grammars

We defer the discussion of how shift/reduce parser for a very general class of languages can be built (in a rather complex way) and first illustrate how this can be done by hand in a special case.

**Operator grammars** are grammars

- without  $\epsilon$  production
- in which no production RHS has two adjacent non-terminals

This class of grammars is interesting because it captures arithmetic expressions (which are difficult to handle with other simpler parsing techniques).

Example

$$\begin{aligned}
 E &\rightarrow EAE \mid (E) \mid id \\
 A &\rightarrow + \mid - \mid * \mid / \mid \uparrow
 \end{aligned}$$

is not an operator grammar. However, we can easily transform it into one:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$$

Historically, many parsers have been built on recursive descent techniques employing operator grammar techniques in sub-parser for expressions.

### Operator Precedence Relations

To implement an operator precedence (shift-reduce) parser we introduce so-called **precedence relations** that help us to keep track of handles.

Precedence relations are defined on pairs of terminals

Relation	Interpretation
$a \triangleleft b$	$a$ yields precedence to $b$
$a \doteq b$	$a$ and $b$ have same precedence
$a \triangleright b$	$a$ takes precedence over $b$

To decide when to shift and when to reduce in operator precedence parsing we remove all non-terminal symbols from the sentential form that we want to reduce and insert the proper precedence relations.

We then shift/reduce using the following steps:

1. Scan the string from the left until the first  $\triangleright$  is encountered.
2. Scan backwards skipping all  $\doteq$  relations until the first  $\triangleleft$  is encountered.
3. Everything between the two relation symbols found in this way is the handle to be reduced. This includes, of course, all the non-terminals between the symbols as well as potentially the surrounding non-terminals.

Intuitively we could say that the precedence relations recover the proper parentheses for the expression so that it can properly be parsed.

### Operator Precedence Example

Consider the grammar

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

A useful precedence table is

	<i>id</i>	+	*	\$
<i>id</i>		▷	▷	▷
+	◁		◁	▷
*	◁	◁		▷
\$	◁	◁	◁	

Note: We need to use \$ as the delimiter of the sentential form.

The sentential form

*id* + *id* \* *id* will be transformed into

$$\$ \triangleleft id \triangleright + \triangleleft id \triangleright * \triangleleft id \triangleright \$$$

So the leftmost *id* will be reduced first, followed by the other *ids*. After this we obtain

*E* + *E* \* *E* from which we drop the non-terminals and insert relations. This yields

$$\$ \triangleleft + \triangleleft * \triangleright \$ \text{ which gives us the handle } E * E.$$

### Constructing Precedence Tables

Obviously, the magic lies in the construction of the operator precedence table.

The following heuristics helps if we have an arithmetic expression language with well-defined precedence and associativity rules.

- for two operators  $\phi_1, \phi_2$  if  $\phi_1$  has higher precedence than  $\phi_2$  make  $\phi_1 \triangleright \phi_2$  and  $\phi_2 \triangleleft \phi_1$ . for example  $* \triangleright +, + \triangleleft *$ . In this way the handle belonging to the innermost level of “parentheses” will be selected first.
- for two operators  $\phi_1, \phi_2$  if  $\phi_1$  has the same precedence as  $\phi_2$ :
  - if  $\phi_1, \phi_2$  are left-associative, make  $\phi_1 \triangleright \phi_2, \phi_2 \triangleright \phi_1$ . For example,  $+ \triangleright +, + \triangleright -, - \triangleright -, - \triangleright +$ .
  - if  $\phi_1, \phi_2$  are right-associative, make  $\phi_1 \triangleleft \phi_2, \phi_2 \triangleleft \phi_1$ .

In this way the leftmost/rightmost subexpression of equal precedence will be selected first.

- for all operators  $\phi$  make  $\phi \triangleleft id, id \triangleright \phi, \phi \triangleleft (, ( \triangleleft \phi, ) \triangleright \phi, \phi \triangleright \$, \$ \triangleleft \phi$ . This forces the handle between the \$ end markers (and makes sure that identifier and expressions in parentheses are reduced first).

### Operator Precedence Parsing Algorithm

```

initialize the stack to $;
initialize the input buffer to w$;

repeat forever
  if ($ is on top of the stack and ip point to $)
    then return;
  else begin
    let a be the topmost terminal on the stack
    let b be the symbol that ip points to;

    if a < b or a = b then begin (* shift *)
      push b onto the stack;
      advance ip to next input symbol;
    end;
    else if a > b then (* reduce *)
      repeat
        x := pop();
      until top() < x
    else error();
  end;
end;
```

Notation: in the algorithm we have used < for  $\triangleleft$ , > for  $\triangleright$ , = for  $\doteq$ .

### Unary Operators in Precedence Parsing

Unary operators usually present a little bit more of a challenge.

For a “special” unary operator (for example, the logical negation  $\neg$ ) we can define the precedence relatively easily:

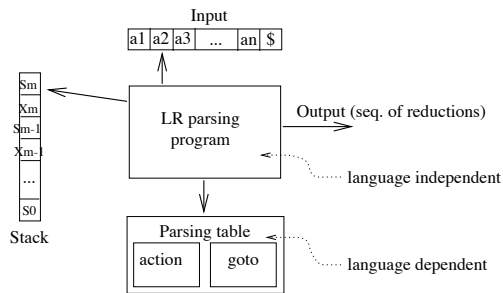
- for any operator  $\phi$  make  $\phi \triangleleft \neg$
- for operators of higher precedence than  $\neg$  make  $\neg \triangleleft \phi$
- for operators of lower precedence than  $\neg$  make  $\neg \triangleright \phi$

Note that the situation becomes more challenging if we have operators that are used both as unary and binary operators, for example “.” or “not”.

## 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**  $error()$

### Example of LR Parsing

Recall the grammar

$exp \rightarrow exp + term \quad (1)$

$exp \rightarrow term \quad (2)$

$term \rightarrow term * factor \quad (3)$

$term \rightarrow factor \quad (4)$

$factor \rightarrow (exp) \quad (5)$

$factor \rightarrow \mathbf{int} \quad (6)$

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.

### Example of LR Parsing (Cont.)

STACK	INPUT	ACTION
0	$int_1 + int_2 \$$	

### Classes of LR Parsers

There are several different classes of LR grammars

- Canonical LR  
largest class of LR grammars  
large number of states, expensive construction  
stops immediately at error position without further reduction
- SLR (simple LR)  
smallest class of LR grammars  
small number of states, simple construction  
makes unnecessary reduce moves when error is encountered
- LALR (lookahead LR)  
more powerful than SLR, but less than canonical LR  
approximately same table size as SLR  
intermediate construction complexity  
employed in real parser generators like YACC.

The basic idea for all these methods is to construct a goto table that models the transition function of a DFA recognizing the viable prefixes of the grammar.

### Summary

We have looked at bottom-up parsing:

- Table driven bottom-up parsing.
- Operator-precedence parsing.
- LR parsing.

### Homework

- Read Chapter 4 of Aho et al.

### Viable Prefixes

**Handles** can only appear on top of the stack in SR parsing.

A prefix of a right sentential form that can appear on top of the stack of an SR parser is called a **viable prefix**.

This is equivalent to the following definitions:

- A prefix of a right sentential form  $w$  is called a viable prefix, if it does not extend past the right hand of the rightmost handle of  $w$ .
- If  $S \xrightarrow{*}_{rrm} uXw \xrightarrow{rrm} uvw$  then  $v$  is a handle of  $uvw$  and each prefix of  $uv$  is a viable prefix.

Therefore, as long as the input seen so far can be reduced to a viable prefix, no error has occurred.

<p style="text-align: center;"><b>Programming Language Implementation</b> <b>VIII</b></p>
---

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

*Reminder:* The basic idea for all LR parser methods is to excute an automaton that recognizes the viable prefixes of the grammar.

A **viable prefix** is a prefix of a right sentential form that can appear on top of the stack of an SR parser.

This is equivalent to the following definitions:

- A prefix of a right sentential form  $w$  is called a viable prefix, if it does not extend past the right hand of the rightmost handle of  $w$ .
- If  $S \xleftarrow{*rrm} uXw \xleftarrow{rrm} uvw$  then  $v$  is a handle of  $uvw$  and each prefix of  $uv$  is a viable prefix.

In the last lecture we looked at LR parsers. In this lecture we will construct the **LR parsing table**.

Among the three classes of LR 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**.

## Items

Construction of the parsing table rquires 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.

Intuitively, the marker indicates which portion of the RHS we have already read and which portion we expect to find next.

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

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

A 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  $\rightarrow$  DFA subset conversion, and the states of the corresponding NFA would be marked by single items.

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

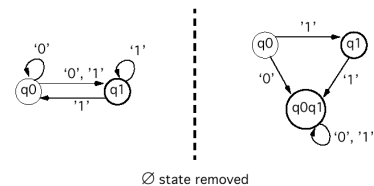
The construction of the automaton for the viable prefixes is similar to how we turn a NFA into a DFA.

## Reminder: NFA-DFA Equivalence

A finite state automata is **deterministic** if it has no transitions via  $\epsilon$  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 non-deterministic state automaton (NFA) can be transformed into an equivalent deterministic state automaton (DFA) such that both automata accept the same language.

The “trick” is to model all possible state combinations (in the NFA) explicitly as separate states (in the DFA). This is called the *subset construction*.



The obvious problem is that this leads to a combinatorial explosion in the number of states.

## LR construction

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

We will need two functions to perform this:

- *closure* (on items) and
- *goto* (the transition function of this automaton)

We start with an augmented grammar  $G'$  which is  $G$  with a new start symbol  $S'$  and the new production  $S' \rightarrow S$ , where  $S$  is the start symbol of the original grammar.

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

## 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  $X \rightarrow Y.Z$  is *valid* for a viable prefix  $vY$  if there is a rightmost derivation  $S \xleftarrow{*}_{rrm} vXw \xleftarrow{*}_{rrm} vYZw$ .

Informally this means: If we find  $vY$  on top of the stack and  $X \rightarrow Y.Z$  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).

$goto(I, X)$  where  $I$  is the set of all valid items for a viable prefix  $w$  is the set of all valid items for the viable prefix  $wX$ .

This will be the state transition function of the automaton.

## Closure

First we need to define the closure operation on items.

The **closure** of a set of items  $I$ , written  $closure(I)$ , 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  $closure(I)$  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  $\gamma$ .

Consider the augmented grammar

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

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

## 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  $X$ .

For each  $A \rightarrow \alpha \cdot X\beta$  in  $I$  we produce

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

Now we take the closure of these.

Consider the grammar

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

If  $I$  is

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

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

### The Canonical Collection of LR(0) items

To build the parsing table we must first compute the **collection of sets of LR(0) items**,  $C$ , for the augmented grammar.

- 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

```

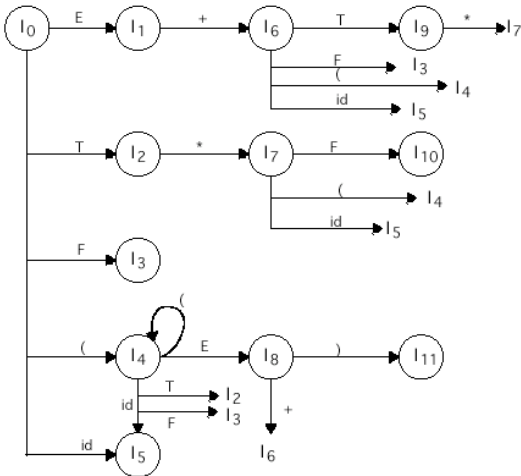
exp' → exp
exp  → exp + term
exp  → term
term → term * factor
term → factor
factor → (exp)
factor → int
    
```

we have the collection of sets of items:

$I_0: exp' \rightarrow \cdot exp$	$I_5: factor \rightarrow \cdot int$
$exp \rightarrow \cdot exp + term$	
$exp \rightarrow \cdot term$	$I_6: exp \rightarrow exp + \cdot term$
$term \rightarrow \cdot term * factor$	$term \rightarrow \cdot term * factor$
$term \rightarrow \cdot factor$	$term \rightarrow \cdot factor$
$factor \rightarrow \cdot (exp)$	$factor \rightarrow \cdot (exp)$
$factor \rightarrow \cdot int$	$factor \rightarrow \cdot int$
$I_1: exp' \rightarrow exp \cdot$	$I_7: term \rightarrow term * \cdot factor$
$exp \rightarrow exp \cdot + term$	$factor \rightarrow \cdot (exp)$
	$factor \rightarrow \cdot int$
$I_2: exp \rightarrow term \cdot$	$I_8: factor \rightarrow (exp) \cdot$
$term \rightarrow term \cdot * factor$	$exp \rightarrow exp \cdot + term$
$I_3: term \rightarrow factor \cdot$	
	$I_9: exp \rightarrow exp + term \cdot$
$I_4: factor \rightarrow (\cdot exp)$	$term \rightarrow term \cdot * factor$
$exp \rightarrow \cdot exp + term$	
$exp \rightarrow \cdot term$	$I_{10}: term \rightarrow term * factor \cdot$
$term \rightarrow \cdot term * factor$	
$term \rightarrow \cdot factor$	$I_{11}: factor \rightarrow (exp) \cdot$
$factor \rightarrow \cdot (exp)$	
$factor \rightarrow \cdot int$	

### The Characteristic Automaton

The previous grammar corresponds to the characteristic automaton below. Its states represent the canonical LR(0) item collection and the state transition function is derived from the *goto* function from above.



Abbreviations used:  $E$  is  $Exp$ ,  $T$  is  $term$ ,  $F$  is  $factor$ .

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

If this automaton (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 set  $I_i$  of items in  $C$ :
  1. If  $A \rightarrow \alpha \cdot a \beta$  is in  $I_i$  and  $goto(I_i, a) = I_j$  then set  $action[s_i, a]$  to *shift*  $s_j$ . Note that  $a$  must be a terminal.
  2. If  $A \rightarrow \alpha \cdot$  is in  $I_i$  then set  $action[s_i, a]$  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  $action[s_i, \$]$  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*.

Let the initial parser state be the state derived from the item set  $I_i$  that contains  $S' \rightarrow S$ .

If this method produces conflicts, the grammar is not SLR(1).

### SLR(1) Table Example

Our example expression grammar

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

State	Action				Goto		
	id	+	(	\$	E	T	F
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		

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  $action[s_1, +] = shift(s_6)$ .

$I_2$  generates  $action[s_2, \$] = action[s_2, +] = action[s_2, (] = reduce(E \rightarrow T)$  since  $follow(E) = \{ \$, +, ( \}$  and  $action[s_2, *] = shift(s_7)$ .

etc.

### SLR(1) Grammars and Conflicts

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

The following grammar, which models a C assignment, 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

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

The first item in  $I_2$  sets  $action[s_2, =]$  to *shift*. The second item in  $I_5$  sets  $action[s_5, =]$  to *reduce*, since  $=$  is in  $follow(R)$  should be.

This is an example of a *shift/reduce conflict*. Note that the grammar is not ambiguous.

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).

## Summary

We have continued looked at bottom-up parsing:

- Constructing the LR parsing table.

## Homework

- Read Chapter 4 of Aho et al.

## Programming Language Implementation IX

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

- **Generic bottom-up parsing.**
- **Parser Generators (ML-YACC).**

Material for ML-YACC can be found in Appel, Chapter 3 and at <http://www.cs.princeton.edu/~appel/modern/ml/>.

**The material on ML-YACC is mainly for self-guided study and is not examinable.**

## Generic Bottom-Up Parsing

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**).

The first step is to transform the grammar into **Chomsky normal form (CNF)**. This requires the grammar to be  
–  $\epsilon$ -free  
– and each (non  $\epsilon$  production) is of the form  $A \rightarrow a$  or  $A \rightarrow BC$  where  $a$  is a terminal and  $A, B, C$  are non-terminals.

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

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

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

- Initially for  $i = 1, \dots, n$ ,  $T[i, i]$  is  $\{A | A \Rightarrow a_i\}$ .  
From now on we are only concerned with productions of the form  $A \rightarrow BC$ .
- Then we iteratively compute column  $j$  in terms of lower columns.  
For each production  $A \rightarrow BC$ , if for some  $i, k < j$  we have  $B \in T[i, k]$  and  $C \in T[k + 1, j]$  then add  $A$  to  $T[i, j]$ .
- $w$  is in the language iff  $S \in T[1, n]$ .

## Converting to Chomsky Normal Form

To convert a grammar into chomsky normal form we need to perform two steps:

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

Reminder: to make a grammar  $\epsilon$ -free

- determine all non-terminals  $X$  for which  $X \Rightarrow^* \epsilon$ . We call these non-terminals *nullable*.
- replace each production  $p$  of the form  $A \rightarrow B_1, B_2, \dots, B_n$  by a set of productions that is composed of copies of  $p$  with each possible combination of nullable non-terminals removed on the LHS.

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

This can be simply done by adding new non-terminals.

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  $\alpha$  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?

### Simple Implementation of CYK

```
for j := 1 to n do
  T[j,j] := {A | A ⇒ ai}
  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 in G then
        add A to T[i, j]
      end
    end
  end
end
accept if S ∈ T[1, n].
end.
```

### Example of CYK Parsing

Consider the grammar

$$exp \rightarrow exp + exp \mid \mathbf{int}$$

Give a Chomsky normal form grammar for it:

Now parse the string:  $\mathbf{int}_1 + \mathbf{int}_2 + \mathbf{int}_3$ .

### Parser Generators – YACC

Creating LR parsers by hand is quite tedious and error prone, because the table construction is too complicated. Instead we can use a parser generator. The best known parser generator is the unix program **yacc** ("Yet Another Compiler Compiler").

- **bison**, is the yacc implementation as part of the Open Software Foundation's GNU system. It interfaces with **FLEX** and "C" code.
- **ML-YACC** is an ML implementation of YACC (with slight modifications). It interfaces with ML code and ML-Lex.

A parser generator takes

- an attributed LR grammar
- additional code annotations for semantic actions
- (optional) precedence rules to resolve shift/reduce conflicts
- (optional) additional error handling code

It produces a complete LR parser for the given language.

## ML-YACC specification

An ML-YACC specification has the form

```

{user declarations}
%%
{ML-Yacc declarations}
%%
{rules}

```

As in ML-Lex, the “user declarations” contain arbitrary ML code that can later be used in the semantic actions of the grammar.

The “ML-Yacc declarations” contain in particular the declaration of terminal and non-terminals symbols and types. Additional precedence declarations and error handling declarations also go into this part.

Finally, the “rules” form the core of the parser specification. Each rule is a production of the form

```
NT : LHS1 ... LHSn {semantic action code}
```

where *NT* is the RHS non-terminal, *LHS<sub>i</sub>* are the terminals and non-terminals on the left-hand side and the semantic action is given as ML-Code. The code for this action will be executed immediately when this production is used for a reduction.

Alternative right-hand sides are separated by bars.

For example, our usual expression production is

```
exp: exp PLUS exp | exp TIMES exp
```

## Synthesized versus Inherited Attributes

Recall that a *synthesized attribute* is an assignment to an attribute of the LHS symbol (computed from RHS attributes).

An *inherited attribute* is an assignment to one of the RHS symbols (computed from LHS attributes and other RHS attributes).

ML-Yacc only uses a synthesized attribute schema, as for example in

```

EXP : NUM          (NUM)
    | ID           (lookup ID)
    | EXP PLUS EXP (EXP1+EXP2)
    | EXP TIMES EXP (EXP1*EXP2)
    | EXP DIV EXP  (EXP1 div EXP2)
    | EXP SUB EXP  (EXP1-EXP2)

```

This means that *inherited attributes have to be simulated* by returning a function instead of a value from the derived symbol.

This function takes the value of the intended inherited attribute and computed the value for the synthesized attribute.

## Semantic Actions

Each non-terminal symbol can have a value associated with it. This value is set by the semantic action of the production, ie. the value to which the semantic action evaluates is assigned to the value of this non-terminal. Of course, the types must be in agreement.

Whether a grammar symbol has a value is derived from its declaration in the “ML-YACC declarations” section, for example

```
%nonterm EXP of int
```

declares the non-terminal *EXP* to have a value of type integer.

**Note that you can only use non-polymorphic types.**

This means, of course, that every symbol can only have a single value and that *multiple attributes for a symbol must be simulated with a tuple or other datatype.*

If a symbol has no value, its actions are still evaluated for **side-effects** and the returned values are ignored.

The values of the non-terminals on the left-hand side are accessible as *symbol<sub>n</sub>* where *n* is the occurrence count of the symbol on the LHS. For example, **exp1** is the value for the first *exp* symbol on the LHS.

## Converting Inherited Attributes

Consider this fragment of our expression attribute grammar:

```

term → factor term'
      term'.v1 := factor.v; term.v := term'.v;
term' → *term
       term'.v := term'.v1 * term.v;
term' → ε
       term'.v := term'.v1;
factor → int
        factor.v := int.v;

```

To simulate its function with synthesized (function) attributes only convert it to:

```

term → factor term'
      term.val := term'.fn(factor.val);
term' → *term
       term'.fn := (fn t => t * term.val);
term' → ε
       term'.v := fn t => t;
factor → int
        factor.val := int.val;

```

where *term'.fn* is a synthesized function attribute.

### A first Example

The following code is a complete parser (and evaluator) for simple arithmetic expressions. It parses appropriately lex'ed arithmetic expressions and returns their value. If an expression is prefixed with a `PRINT` token (statement) its value will additionally be printed to the console.

```
fun lookup "x" = 5
  | lookup "y" = 6
  | lookup "z" = 7
  | lookup s = 0

%%

%eop EOF SEMI
%pos int

%left SUB PLUS
%left TIMES DIV
%right CARAT

%term ID of string | NUM of int | PLUS | TIMES | PRINT |
  SEMI | EOF | CARAT | DIV | SUB
%nonterm EXP of int | START of int option

%name Calc

%subst PRINT for ID
%prefer PLUS TIMES DIV SUB
```

### Declarations in Example (cont.)

The declarations of terminals and non-terminals and the grammar rules should be easy enough to understand. What are the remaining declarations.

First note that we have simply defined a *lookup* function to implement a variable valuation, ie. a trivial *symbol table*. In reality this would, of course, not be hard-coded in the grammar.

`%name` gives the parser a name that we need later to couple it with the lexer.

`%eop EOF SEMI` declares that the tokens `EOF` and `SEMI` (semicolon) may follow the start symbol, i.e. end the parse. ML-YACC cannot recognise the end-of-input, so the lexer must insert an appropriate token.

`%noshift EOF` declares that `EOF` may not be shifted. This may seem obvious but must be declared so that the parser does not attempt to shift it in cases of error recovery.

`%pos` defines the (non-polymorphic) ML type for the position attribute of tokens. The position values for the tokens can be used for error messages and can be accessed as *symbolnamen + 1left* and *symbolnamen + 1right*. The position values for terminals must be set by the lexer (see below).

`%left,%right` declare the associativity and order of precedence of symbols in increasing order. Symbols in the same declaration have the same precedence. This is used for conflict resolution (see below).

```
%keyword PRINT SEMI

%noshift EOF
%value ID ("bogus")
%nodefault
%verbose
%%

START : PRINT EXP (print (Int.toString EXP);
  print "\n";
  TextIO.flushOut TextIO.stdOut;
  SOME EXP)
  | EXP (SOME EXP)
  | (NONE)

EXP : NUM (NUM)
  | ID (lookup ID)
  | EXP PLUS EXP (EXP1+EXP2)
  | EXP TIMES EXP (EXP1*EXP2)
  | EXP DIV EXP (EXP1 div EXP2)
  | EXP SUB EXP (EXP1-EXP2)
  | EXP CARAT EXP (let fun e (m,0) = 1
    | e (m,1) = m*e(m,1-1)
    in e (EXP1,EXP2)
    end)
```

You will find this code in the ML-YACC example directory.

Remember that *SOME* and *NONE* are the constructors of the *option* datatype.

`%subst`, `%prefer`, `%keyword`, `%value` declarations are used in error recovery and will be explained later.

`%nodefault`, `%verbose` are used for debugging purposes only.

### Conflicts in ML-YACC

A grammar that is not LR(1) may produce conflicts in the parser table. These could be either

- shift/reduce conflicts, which will by default be resolved in favour of shifting or
- reduce/reduce conflicts, which will by default be resolved in favour of the first production that appears earlier in the grammar.

It is in general not a good idea to rely on these defaults and error messages will be issued if the default is used.

Sometimes the default leads to reasonable actions (and can be accepted after careful analysis). Consider the meanwhile familiar dangling else problem:

```
prog: stmlist ()

stm : ID ASSIGNOP ID      ()
    | WHILE ID DO stm    ()
    | BEGIN stmlist END   ()
    | IF ID THEN stm     ()
    | IF ID THEN stm ELSE stm ()

stmlist: stm              ()
        | stmlist SEMICOLON stm ()
```

### Precedence in ML-YACC

In general the grammar should be re-written such that conflicts are avoided.

In ML-YACC we can also achieve conflict resolution by using **precedence declarations**.

Remember: `%left,%right` declare the associativity and order of precedence of symbols in increasing order. Symbols in the same declaration have the same precedence.

We know that our naive expression grammar is ambiguous:

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

We have previously resolved this by operator precedence parsing (or by rewriting the grammar).

If we declare the proper precedences/associativities in ML-YACC with the declarations

```
%left SUB PLUS
%left TIMES DIV
%right CARAT
```

the conflicts will be resolved appropriately.

Use ML-YACC to produce the parse table for the naive grammar. One of the conflicts will be found in a state with lookahead “+” which has the items

The parsing table for this grammar produces a conflict. If ML-YACC is used with the `%verbose` declaration we will can inspect the state table (produced as output).

We find that in one of the states we have **shift/reduce conflict** with `shift(ELSE)` or reducing with the first “IF” rule.

ML-YACC will by default choose `shift(ELSE)`.

As a consequence the `ELSE` will be bound to the innermost open `THEN`. If this is the intended effect the conflict is acceptable.

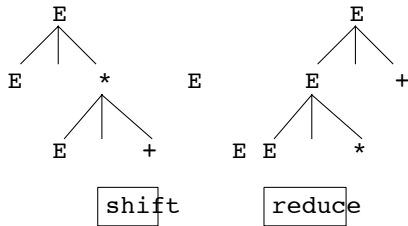
$$\begin{aligned} E &\rightarrow E * E. \quad (\text{lookahead } +) \\ E &\rightarrow E. + E \end{aligned}$$

In this state, the top of the stack will contain  $E * E$ .

- Shifting would lead to  $E * E +$ , then to  $E * E + E$  and subsequently by reduction of  $E + E$  to  $E$  again to  $E * E$ .
- Reducing would lead to  $E$  and subsequently to  $E +$ ,  $E + E$  and then again to  $E$  by reduction of  $E + E$  to  $E$ .

### Conflict Resolution by Precedence

The parse trees for these alternatives are given below



We should therefore **reduce** in favour of the operator with higher precedence.

The case for the conflict with items

$$E \rightarrow E + E. \text{ (lookahead +)}$$

$$E \rightarrow E. + E$$

can be analysed in a similar fashion. *Shifting* will make “+” right-associative, *reducing* will make it left-associative.

To decide which *action* takes precedence in a *shift/reduce* conflict we need to look at the items:

$$E \rightarrow E * E. \text{ (lookahead +)}$$

$$E \rightarrow E. + E$$

*The precedence of a reduce action is given by the precedence of the last token on the RHS of the item, the precedence of a shift action is given by the item to be shifted.*

Thus the precedences are:

- $E \rightarrow E * E$ . (action: reduce) has the precedence of “\*”,
- $E \rightarrow E. + E$  (action: shift) has the precedence of “+”

the rule with the action precedence will be executed (here: reduce).

Accordingly, if the precedence does not resolve the conflict (i.e. is the same), *left associativity* will favour reducing, *right associativity* will favour shifting.

### Error Correction in ML-YACC

Instead of this type of *local recovery* ML-YACC uses a *global recovery* (and more costly) technique called **Burke-Fisher Parsing**.

Imagine the correct ML definition

```
val inc = fn x => 1 + x;
```

Consider the damaged version

```
val inc = x => 1 + x;
```

The error would only be encountered when the => token is read and could therefore not be corrected appropriately by skipping input.

Burke-Fisher Parsing proceeds in the following way: If an error is encountered

- Perform a correction attempt at the current input position and each of the preceding 15 input positions (15 is a heuristic choice from experience).
- At each correction position try to
  - delete the token,
  - substitute the token by some other token,
  - insert an additional token
- Try to continue parsing with this change past the error position.

- Among all possible corrections choose the one that allows to continue parsing furthest past the error point.

Obviously the parser must be able to back up over the last 15 shift operations. To do this ML-YACC maintains two stacks: the *current* stack and the *previous* stack that are joined by a queue.

## Semantic Actions and Error Correction

It is obvious that semantic actions may only be executed during the error-recovery attempts if they are **side-effect free** as their side-effects could not be undone if the recovery attempt is unsuccessful.

ML-Yacc uses higher-order functions to defer the evaluation of all user semantic actions. The semantic actions will only be executed once the corresponding parser action are committed (i.e. reach the *previous* stack).

`%pure` may be used as a declaration if all semantic actions in a grammar are side-effect free (and always terminate). With this declaration the parser will not defer their evaluation.

We can now also understand the other declarations that we have skipped above:

`%prefer` lists the keyword whose insertion should be attempted first.

`%subst` lists pairs *terminal* for *terminal* that specify a heuristics for preferred substitution attempts. The pairs on this list must be separated by bars (“—”).

Finally, each token that is inserted and has a semantic value must, of course, supply a default value. This is handled by the *value declaration*, in our example grammar:

```
%value ID ("dummy")
```

## Combining ML-Lex and ML-YACC

For the example parser we first generate a lexer. Store the following code in a file named `lex-ex.lex`.

```
structure Tokens = Tokens

type pos = int
type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult= (svalue,pos) token

val pos = ref 0
val eof = fn () => Tokens.EOF(!pos,!pos)
val error = fn (e,l : int,_) =>
  TextIO.output(TextIO.stdOut,"line "
    ^ (Int.toString l) ^ ": " ^ e ^ "\n")
%%
%header (functor
%   CalcLexFun(structure Tokens: Calc_TOKENS));
alpha=[A-Za-z];
digit=[0-9];
ws = [\ \t];
%%
\n      => (pos := (!pos) + 1; lex());
{ws}+  => (lex());
{digit}+ => (Tokens.NUM
  (foldl (fn (a,r) =>
    ord(a)-ord("#0")+10*r) 0
    (explode yytext) ),
```

## Generating a Parser with ML Yacc

(For details you can also read the *ML-YACC* documentation, from which the following is adapted, but please make sure to follow the instructions here, as there are some bugs in the official documentation).

To generate a parser using ML-YACC you need to follow these steps:

1. Run ML-Lex to create the lexical analyzer
2. Run ML-Yacc on the specification file for a grammar
3. Load the ML-Yacc libraries
4. Load the .sig file that ML-Yacc produced
5. Load the lexer that ML-Lex produced
6. Load the .sml file that ML-Yacc produced
7. Join the parser structure by applying functors
8. Define functions that couple lexer and parser

```
!pos,!pos));
"+"    => (Tokens.PLUS(!pos,!pos));
"*"    => (Tokens.TIMES(!pos,!pos));
";"    => (Tokens.SEMI(!pos,!pos));
{alpha}+ => (if yytext="print"
             then Tokens.PRINT(!pos,!pos)
             else Tokens.ID(yytext,!pos,!pos)
            );
"- "   => (Tokens.SUB(!pos,!pos));
"^"    => (Tokens.CARAT(!pos,!pos));
"/"    => (Tokens.DIV(!pos,!pos));
"."    => (error ("ignoring bad character "^yytext,
!pos,!pos);
lex());
```

Note that the lexer contains a declaration for a *pos* type (token position).

The parser generator will generate a structure *Tokens* that the lexer uses. The declarations

```

type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult = (svalue,pos) token

```

are mandatory and create the attribute types for the terminals.

The declaration

```

%header (functor
    CalcLexFun(structure Tokens: Calc_TOKENS));

```

is also mandatory. It causes ML-Lex to create a functor for a lexer. The string *Calc* in this declaration refers to the name of the parser (given in the `%name` declaration of the parser) and has to be substituted if your parser has any other name.

### Using the Parser

Now restart SML and load all the libraries for the parser as well as the files that were generated by ML-Lex and ML-YACC.

```

Standard ML of New Jersey, Version 110.0.7, ...
- use "/local/lib/sml/src/ml-yacc/lib/base.sig";
...
- use "/local/lib/sml/src/ml-yacc/lib/join.sml";
...
- use "/local/lib/sml/src/ml-yacc/lib/lrtable.sml";
...
- use "/local/lib/sml/src/ml-yacc/lib/stream.sml";
...
- use "/local/lib/sml/src/ml-yacc/lib/parser2.sml";
...
- use "yacc-ex.grm.sig";
...
- use "lex-ex.lex.sml";
...
- use "yacc-ex.grm.sml";

```

### Combining ML-Lex and ML-YACC (cont.)

After you have generated the file for the lexer, store the example ML-YACC grammar from above in a file called `yacc-ex.grm`.

Now create a file "sources.sml" (don't change the name) with the following contents:

```

Group is
    ml-yacc-lib.cm
    lex-ex.lex
    yacc-ex.grm

```

This file will be used by the so-called compilation manager, the ML equivalent of the Unix command "make". We will not go into details of CM, but we need it to invoke ML-YACC.

Start SML and invoke the compilation manager:

```

bruce_39% sml
Standard ML of New Jersey, Version 110.0.7, ...
- CM.make();
[starting dependency analysis]
[scanning sources.cm]
...

```

This will load the YACC libraries and invoke the ML-Lex and ML-YACC on your specifications. Afterwards you will find the compiled `.sig` and `.sml` files in your directories (plus a `.desc` file with the verbose output of ML-YACC).

### Using the Parser (cont.)

Next, you must create the signature for the parser:

```

- structure CalcLrVals =
= CalcLrValsFun(structure Token = LrParser.Token);
structure CalcLrVals :
    sig
        structure ParserData : <sig>
            structure Tokens : <sig>
                end
    end
- structure CalcLex =
= CalcLexFun(structure Tokens = CalcLrVals.Tokens);
structure CalcLex :
    sig
        structure Internal : <sig>
            structure UserDeclarations : <sig>
                exception LexError
                val makeLexer : (int -> string)
                    -> unit -> Internal.result
            end
    end
- structure CalcParser=
= Join(structure ParserData = CalcLrVals.ParserData
= structure Lex = CalcLex
= structure LrParser = LrParser );
structure CalcParser : PARSE

```

This ultimately generates a structure `PARSER` that contains the finished Parser. You now need to define functions that invoke this parser and couple it with the lexer.

### Using the Parser (cont.)

We define a function *invoke* that calls the parser with a function for error output.

```
fun invoke lexstream =
  let fun print_error (s,i:int,_) =
        TextIO.output(TextIO.stdOut,
                      "Error, line " ^
                      (Int.toString i) ^ ", "
                      ^ s ^ "\n")
      in CalcParser.parse(0,lexstream,print_error,())
      end
end
```

### Using the Parser (cont.)

Finally, the function *parse* couples the lexer and the parser:

```
fun parse () =
  let val lexer = CalcParser.makeLexer
      (fn _ =>
       TextIO.inputLine TextIO.stdIn)
      val dummyEOF = CalcLrVals.Tokens.EOF(0,0)
      val dummySEMI = CalcLrVals.Tokens.SEMI(0,0)
      fun loop lexer =
          let val (result,lexer) = invoke lexer
              val (nextToken,lexer) =
                  CalcParser.Stream.get lexer
              in case result
                  of SOME r =>
                     TextIO.output(TextIO.stdOut,
                                     "result = "
                                     ^ (Int.toString r) ^ "\n")
                  | NONE => ();
                  if CalcParser.sameToken(nextToken,
                                           dummyEOF) then ()
                  else loop lexer
              end
          in loop lexer
          end
  end
```

Note that the lexer now returns a stream instead of a sequence of tokens.

### Using the Parser (cont.)

The parser is now ready to be used

```
- parse ();
3+4*5;
result = 23
print 8+16*2;
40
result = 40
```

### Summary

We have looked at:

- Generic bottom-up parsing (CYK).
- The ML-YACC parser generator.

### Homework

- Read Section 4.4 of Aho et al.
- Read Section 3.4 and 3.5 of Appel
- Study the ML-YACC documentation at <http://www.cs.princeton.edu/~appel/modern/ml/ml-yacc/>. (but be aware that there are mistakes in it!)
- Modify the expression parser from the ML-YACC given in the lecture such that it accepts and evaluates function symbols (such as *sin(X)*) and unary minus without brackets, i.e. expressions of the form *PRINT*3 + 4 \* -5.

# Programming Language Implementation X

In this lecture we will look at **code generation** and in particular at

- **runtime environment**
- **intermediate code generation**
- **register allocation**

The material is (loosely) based on Aho et al Chapters 7, 8 and 9.

## Run-time environment

To understand code generation one needs to understand what should happen at run-time. In particular we need to understand **allocation** and **deallocation** of data objects. This is managed by the **run-time support** package.

The design of the **run-time support** package is influenced by the HL language it supports.

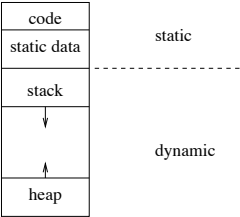
**FORTRAN:** The size of data structures is determined at compile time and sub-programs cannot be recursive.

**Pascal family:** Storage may be allocated dynamically and procedures can be passed as parameters and called recursively.

**Functional and Logic languages:** Allocation and deallocation of storage is invisible to the programmer.

We shall focus on the Pascal family.

## Activation Records



Memory is split into

- **program code**
- **static data**
- **stack**
- **heap.**

An **activation record** is pushed on to the stack when a procedure is called and popped off when control returns from the procedure.

The size of the activation record depends on the procedure's parameters and local variables. It may not be known at compile-time. **Why?**

## Activation Records (Cont)

result if function
dynamic link
static link
return address
saved environment
parameters
local variables
temporary variables

The activation record might contain:

- A location for the result in the case it is a function.
- A pointer to the stack frame of the calling procedure (ie the **dynamic predecessor**).
- A pointer to the stack frame of the textually surrounding procedure (ie the **static predecessor**).
- The return address for the calling procedure.
- Environment information such as register values which need to be restored on return.
- Memory locations for the parameters.
- Memory locations for the local variables.
- Memory locations for the temporary variables generated in expression evaluation.

Normally we keep the **stack top (ST)** and the **local base (LB)** in registers.

## Procedure Invocation

**Calling** a procedure/function

- Evaluates arguments and assigns values to the parameters.
- Sets the **link** data (ie dynamic and static predecessor and the return address).
- Jumps to procedure entry (found in some static table).

**Returning** from a procedure/function

- Restore the calling environment.
- Jump to the return address.
- If there is a return value, copy to temporary variable.

## Static Links

The obvious question is why do we need a **static link** as well as a **dynamic link**?

Give the stack for the Cascal program:

```

program h {
  int i, j;

  int function p {
    int x;

    int function r {
      int y;
      j := j+1;
    }

    if i>0 then {
      i:= i-1;
      p();
      pr: }
    else {
      r();
      rr: }
  }

  i := 2;
  j := 1;
  p();
  p2r: }
  
```

## Static Links (Cont.)

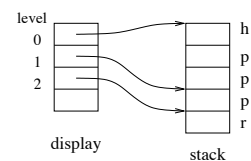
The **textual level** of a procedure (block) is the number of procedures surrounding it.

In a Pascal like language a procedure at level  $K$  can only call procedure at level  $\leq K + 1$ .

To access a variable at offset  $x$  level  $j$  from a procedure at level  $i$  we follow  $i - j$  static links up the stack to find the right activation record and then go to offset  $x$ .

**Exercise:** How do you compute the value of the static link in the procedure being called?

## Displays



We can avoid the unravelling of static links by maintaining an array of registers indexed by textual level which point to the appropriate activation record.

This is called a **display**.

If  $i$  is the level of the called procedure we need to set  $display[i]$  on call and restore its value on return from the call.

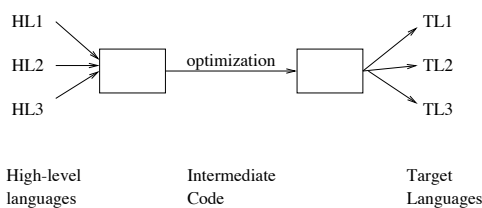
## Code Generation

The aim is to produce code which is:

- **efficient**
- **compact**
- uses **registers** wisely
- is **correct**.

Typical steps in code generation are:

- **Intermediate code generation**
- **Optimization** (this is optional)
- **Target code generation**



Using a machine-independent intermediate language is good because:

- **retargeting** is facilitated
- facilitates machine-independent **code optimization**

## Intermediate Code

One common form of intermediate code is **three address** code. This is a sequence of statements of form

```
x := y <op> z
```

Thus a complex source language expression like

```
x := y + u * z
```

will give rise to the statements

```
t1 := u * z
x := y + t1
```

Three address statements are similar to assembly code:

- **Assignment statements** of the above form.
- **Assignment statements** of the form  

```
x := <op> z
```

where **<op>** is a unary operation.
- **Copy statements** of the form  

```
x := z
```

## Abstract Machines / Virtual Machines

Instead of compiling directly into machine code, the target is often a virtual machine.

A virtual machine is essentially an interpreter for a virtual language that runs on the target machine.

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.

As a consequence, translation into virtual machine code is easier. Native machine code is more complex, 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

- higher portability
- security (easier encapsulation)

However, native code is typically much faster.

- **Unconditional jumps** of form

```
goto L
```

- **Conditional jumps** of form

```
if x <relop> y goto L
```

- **Parameter setting** statements and **procedure call** and **return y**.

```
param x1
param x2
...
param xn
call p, n
```

- **Indexed assignments** of form

```
x := y[i]
y[i] := x
```

- **Address and pointer assignments** of form

```
x := &y
x := *y
*x := y
```

This is only one possible intermediate code. See Chapter 8 of Aho et al for more details.

## Intermediate Code Generation

It is straightforward to use attribute grammars to construct intermediate code.

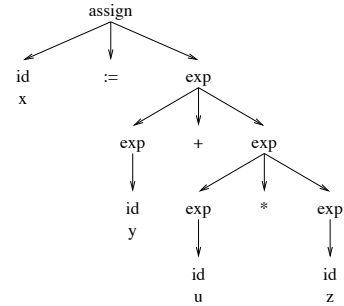
Production	Semantic Rules
$assign \rightarrow id := exp$	$assign.code :=$ $exp.code @ gen(id.place := exp.place)$
$exp_0 \rightarrow exp_1 + exp_2$	$exp_0.place := newtemp$ $exp_0.code :=$ $exp_1.code @ exp_2.code @$ $gen(exp_0.place := exp_1.place + exp_2.place)$
$exp_0 \rightarrow exp_1 * exp_2$	$exp_0.place := newtemp$ $exp_0.code :=$ $exp_1.code @ exp_2.code @$ $gen(exp_0.place := exp_1.place * exp_2.place)$
$exp_0 \rightarrow (exp_1)$	$exp_0.place := exp_1.place$ $exp_0.code := exp_1.code$
$exp_0 \rightarrow id$	$exp_0.place := id.place$ $exp_0.code := nil$

## Intermediate Code Generation (Cont)

For example

$x := y + u * z$

will give rise to:



## Target Code Generation

### Instruction selection

If we do not care about efficiency it is usually easy to generate target code from each three address statement.

Unfortunately we usually do care about efficiency!

### Register allocation

It is better to use register operands. Use of registers involves two steps

- **Register allocation** in which we select the variables to be stored in registers.
- **register assignment** in which we select the specific registers.

We note that optimal assignment is NP-hard.

## Register Allocation

In fact register allocation can be viewed as *k*-graph colouring.

In the graph, variables are nodes and nodes are connected if they are alive at the same time. If we have *k* registers we try to color the graph nodes so that no two connected nodes have the same color using only *k* colours.

Consider the intermediate code

```

<x,y alive>
t1 := x + 1
t2 := y + t1
t3 := y * t2
z := x + t3
<x,z alive>
  
```

What is the register allocation graph?

What is the minimal number of registers needed?

## Peephole Optimization

Peephole optimization shifts an inspection window over the generated code to discover code segments that can be optimized locally. This is normally used for machine-dependant optimization.

Assuming that the target machine has an increment instruction INC working on a memory location, 9-11 can be optimized to the single instruction INC 19.

label	address	instruction	operand	
...				
	3	STORE	19	count
	4	LOADC	1	
	5	STORE	20	result
Label1	6	LOAD	19	count
	7	SUB	21	value
	8	JUMPGE	16	Label2
<hr/>				
	9	LOAD	19	count
	10	ADDC	1	⇒
	11	STORE	19	count
<hr/>				
	12	LOAD	20	result
	13	MUL	19	count

Such code modifications have to be done prior to assembly.

## Summary

We have looked at

- runtime environment
- intermediate code generation
- register allocation
- peephole optimization

## Homework

- Read Chapters 7, 8 and 9 of Aho et al.
- Give attribute rules to generate code for a **while** loop and a procedure call (first give the grammar!).