

# Programming Language Implementation V

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:  $L = \{w cw \mid w \in (a \mid b)^*\}$* , 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  $P$ .

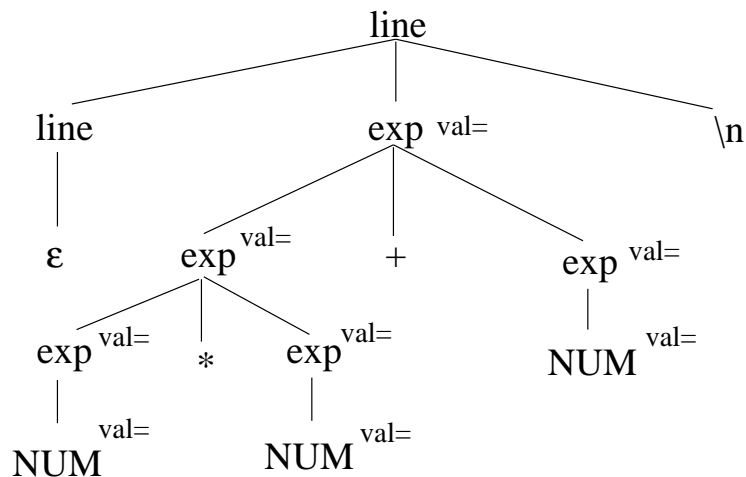
## Attribute Grammar – Synthesized

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:



## 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 $NUM.rep = \text{'words'}$
$NUM \rightarrow 0 \mid 1 \mid \dots \mid 9$	if $NUM.rep = \text{'digits'}$

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

## 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 – 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.

$$\begin{aligned} \text{exp} &\rightarrow \text{term exp}' && \text{exp}' \cdot v1 := \text{term} \cdot v; \text{exp} \cdot v := \text{exp}' \cdot v; \\ \text{exp}' &\rightarrow +\text{exp} && \text{exp}' \cdot v := \text{exp}' \cdot v1 + \text{exp} \cdot v; \\ \text{exp}' &\rightarrow \epsilon && \text{exp}' \cdot v := \text{exp}' \cdot v1; \\ \text{term} &\rightarrow \text{factor term}' && \text{term}' \cdot v1 := \text{factor} \cdot v; \text{term} \cdot v := \text{term}' \cdot v; \\ \text{term}' &\rightarrow * \text{term} && \text{term}' \cdot v := \text{term}' \cdot v1 * \text{term} \cdot v; \\ \text{term}' &\rightarrow \epsilon && \text{term}' \cdot v := \text{term}' \cdot v1; \\ \text{factor} &\rightarrow \mathbf{int} && \text{factor} \cdot v := \text{int} \cdot v; \\ \text{factor} &\rightarrow (\text{exp}) && \text{factor} \cdot v := \text{exp} \cdot v; \end{aligned}$$

## 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.

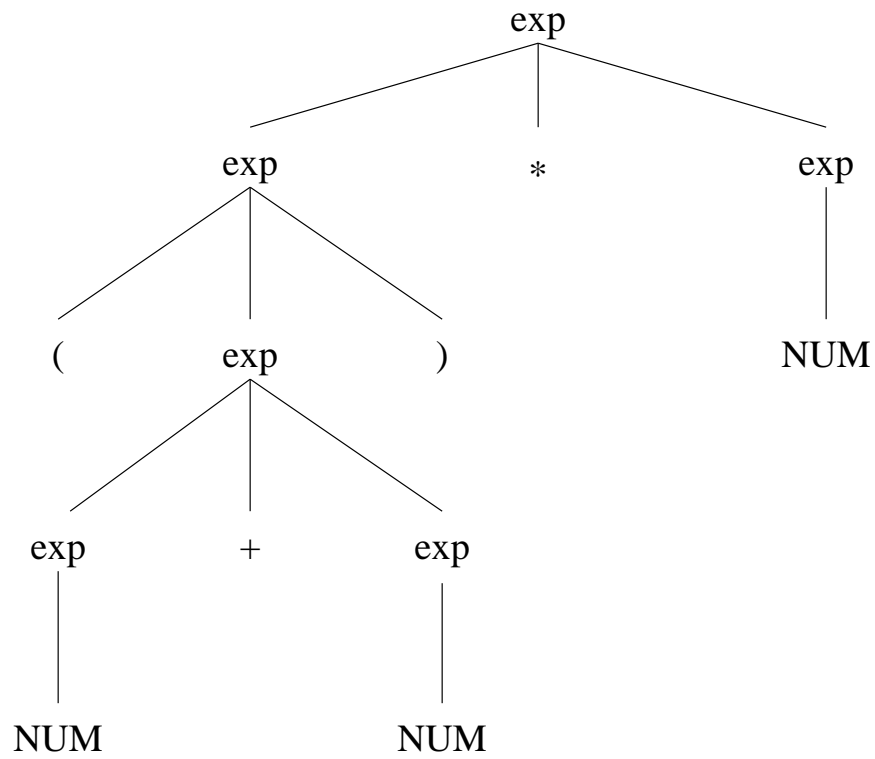
## Construction of Structure Trees

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

$$exp \rightarrow exp - exp,$$

$$exp \rightarrow exp \textit{ div } exp,$$

$$exp \rightarrow \textit{ floor}(exp),$$

where *div* is integer division and *floor* takes a real and returns an integer.

# Programming Language Implementation VI

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

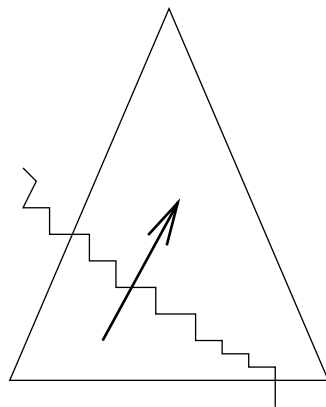
In **bottom-up parsing** the parse tree is built from the leaves upwards interpreting productions from right to left (ie. the input is reduced to the start symbol of  $G$ ).

Example

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

inverted rightmost derivation:

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



bottom-down  
parsing

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**

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

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  $exp + exp * exp$  has two possible handles.

## Stack Implementatation of Shift-Reduce Parsing

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

The parser starts with

STACK	INPUT
\$	$a_1a_2\dots a_n$ \$

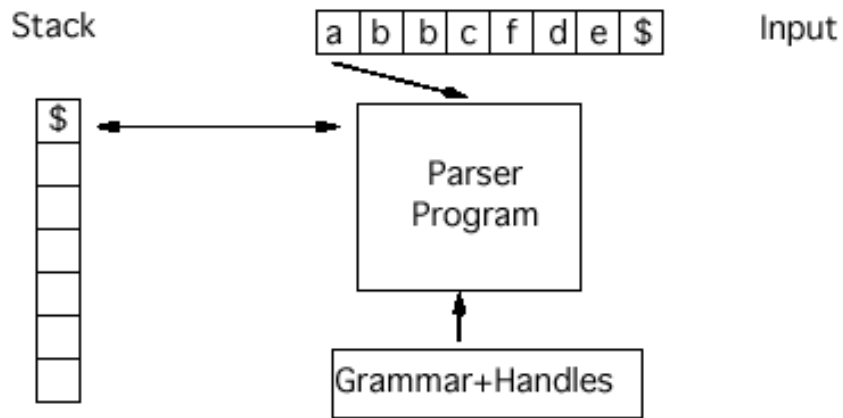
and wants to reach

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=$S or input=$
if stack=$S and input=$ 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 $exp \rightarrow \mathbf{int}$
\$ $exp$	+ $int_2 * int_3$ \$	shift
\$ $exp +$	$int_2 * int_3$ \$	shift
\$ $exp + int_2$	* $int_3$ \$	reduce by $exp \rightarrow \mathbf{int}$
\$ $exp + exp$	* $int_3$ \$	shift
\$ $exp + exp *$	$int_3$ \$	shift
\$ $exp + exp * int_3$	\$	reduce by $exp \rightarrow \mathbf{int}$
\$ $exp + exp * exp$	\$	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.

1. 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.
2. 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.

3. for all operators  $\phi$  make  $\phi \triangleleft id, id \triangleright \phi, \phi \triangleleft (, ( \triangleleft \phi, ) \triangleright \phi, \phi \triangleright ), \phi \triangleright \$, \$ \triangleleft \phi$ . 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:

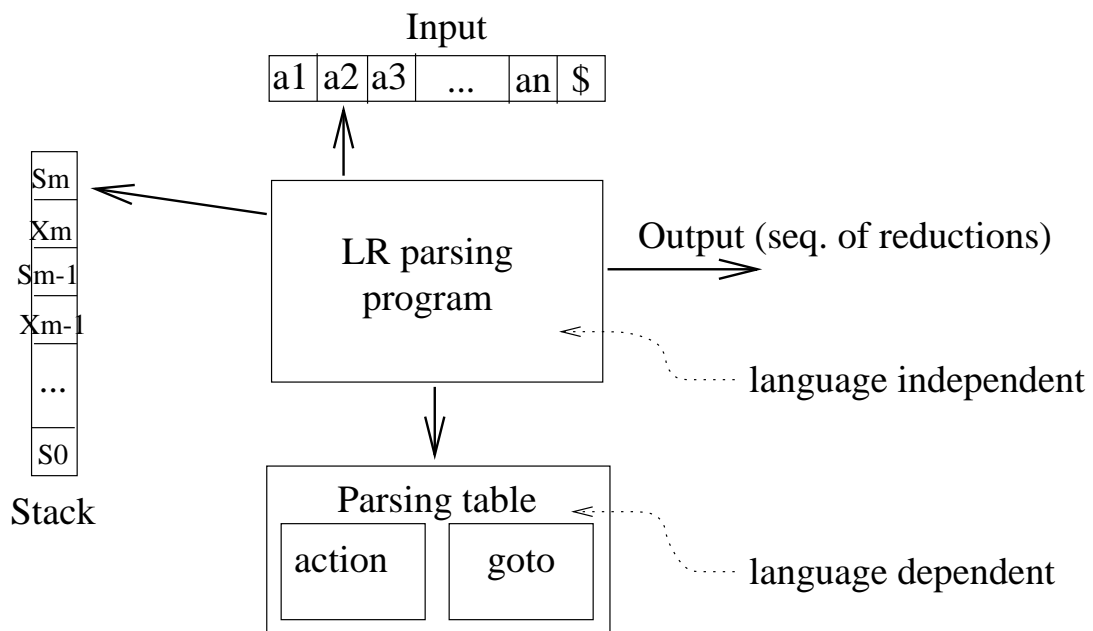
- for any operator  $\phi$  make  $\phi \triangleleft \neg$
- for operators of higher precedence than  $\neg$  make  $\neg \triangleleft \phi$
- for operators of lower precedence than  $\neg$  make  $\neg \triangleright \phi$

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**

$s := \text{top}(\text{stack})$

$a :=$  current input symbol

**if**  $\text{action}[s, a]$  is *shift*  $s'$  **then**

    push  $a$  then  $s'$  on to *stack*

    advance to next input symbol

**else if**  $\text{action}[s, a]$  is *reduce*  $A \rightarrow \beta$  **then**

    pop  $2 \times |\beta|$  symbols off *stack*

$s' := \text{top}(\text{stack})$

    push  $A$  then  $\text{goto}[s', A]$  on to *stack*

    output “ $A \rightarrow \beta$ ”

**else if**  $\text{action}[s, a]$  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

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

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.

## 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  $S \xleftarrow{*}_{rrm} uXw \xleftarrow{rrm} uvw$  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.

## 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.

# Programming Language Implementation VII

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 execute 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**.

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.

## Items

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

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

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.

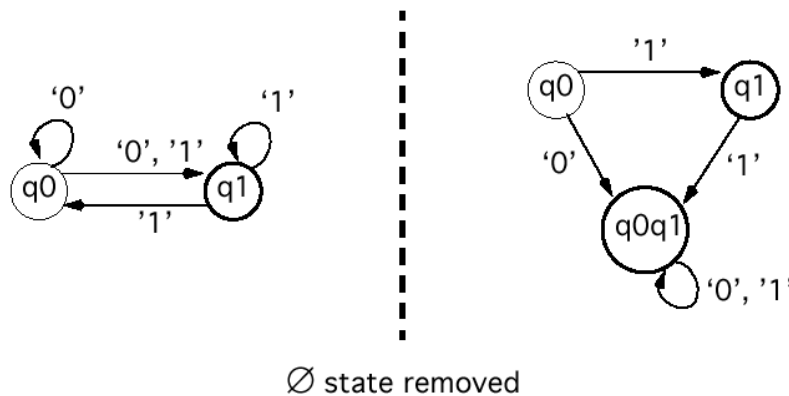
This grouping is essentially an NFA  $\rightarrow$  DFA subset conversion, and the states of the corresponding NFA would be marked by single items.

## 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.

## 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  $closure(I)$ ?

## 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.

## 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

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

we have the collection of sets of items:

$I_0 : \text{exp}' \rightarrow \cdot \text{exp}$   
 $\text{exp} \rightarrow \cdot \text{exp} + \text{term}$   
 $\text{exp} \rightarrow \cdot \text{term}$   
 $\text{term} \rightarrow \cdot \text{term} * \text{factor}$   
 $\text{term} \rightarrow \cdot \text{factor}$   
 $\text{factor} \rightarrow \cdot (\text{exp})$   
 $\text{factor} \rightarrow \cdot \mathbf{int}$

$I_5 : \text{factor} \rightarrow \mathbf{int} \cdot$

$I_6 : \text{exp} \rightarrow \text{exp} + \cdot \text{term}$   
 $\text{term} \rightarrow \cdot \text{term} * \text{factor}$   
 $\text{term} \rightarrow \cdot \text{factor}$   
 $\text{factor} \rightarrow \cdot (\text{exp})$   
 $\text{factor} \rightarrow \cdot \mathbf{int}$

$I_1 : \text{exp}' \rightarrow \text{exp} \cdot$   
 $\text{exp} \rightarrow \text{exp} \cdot + \text{term}$

$I_7 : \text{term} \rightarrow \text{term} * \cdot \text{factor}$   
 $\text{factor} \rightarrow \cdot (\text{exp})$   
 $\text{factor} \rightarrow \cdot \mathbf{int}$

$I_2 : \text{exp} \rightarrow \text{term} \cdot$   
 $\text{term} \rightarrow \text{term} \cdot * \text{factor}$

$I_8 : \text{factor} \rightarrow (\text{exp} \cdot)$   
 $\text{exp} \rightarrow \text{exp} \cdot + \text{term}$

$I_3 : \text{term} \rightarrow \text{factor} \cdot$

$I_9 : \text{exp} \rightarrow \text{exp} + \text{term} \cdot$   
 $\text{term} \rightarrow \text{term} \cdot * \text{factor}$

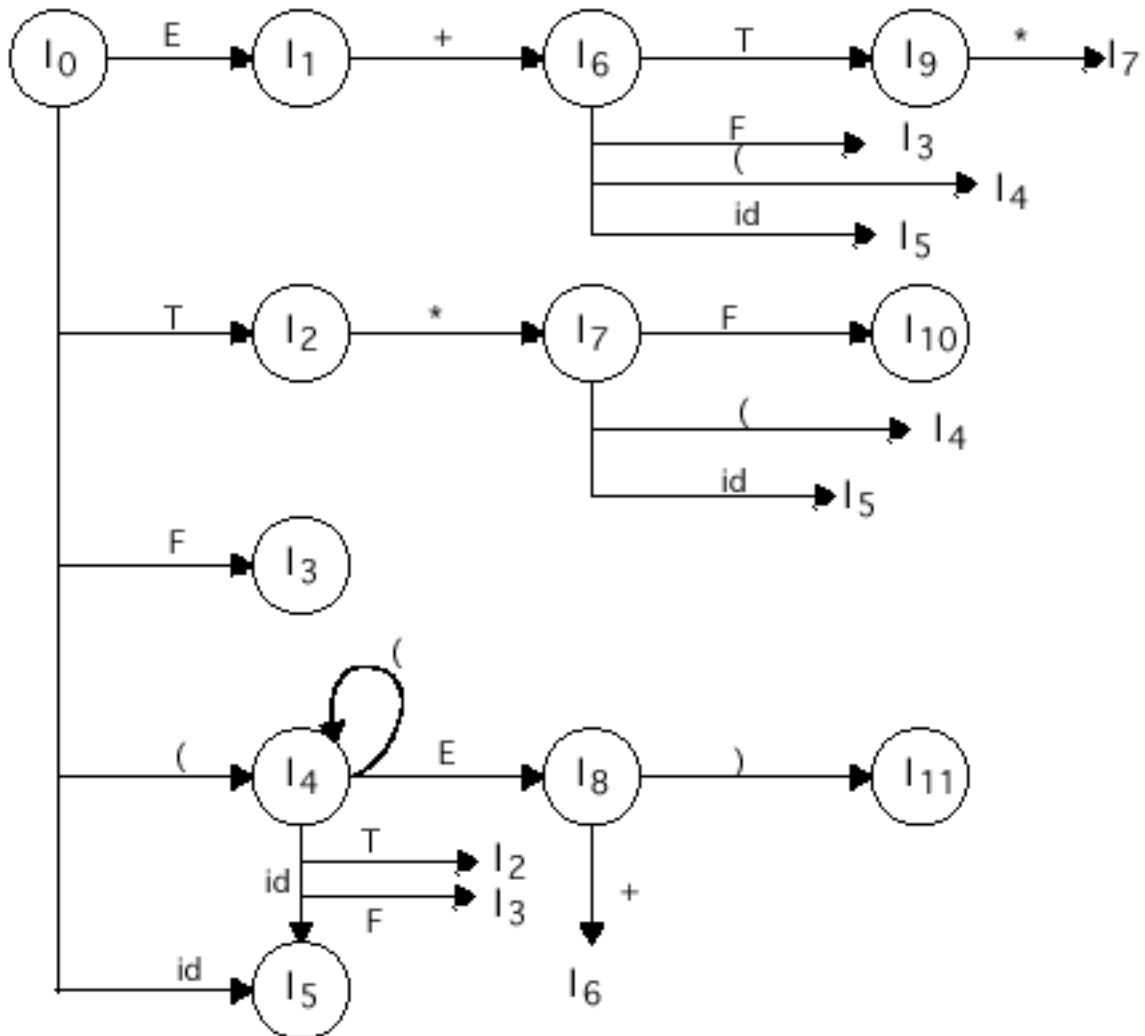
$I_4 : \text{factor} \rightarrow (\cdot \text{exp})$   
 $\text{exp} \rightarrow \cdot \text{exp} + \text{term}$   
 $\text{exp} \rightarrow \cdot \text{term}$   
 $\text{term} \rightarrow \cdot \text{term} * \text{factor}$   
 $\text{term} \rightarrow \cdot \text{factor}$   
 $\text{factor} \rightarrow \cdot (\text{exp})$   
 $\text{factor} \rightarrow \cdot \mathbf{int}$

$I_{10} : \text{term} \rightarrow \text{term} * \text{factor} \cdot$

$I_{11} : \text{factor} \rightarrow (\text{exp}) \cdot$

## 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  $S' \rightarrow .S$ ) 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

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

$$exp \rightarrow term \quad (2)$$

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

$$term \rightarrow factor \quad (4)$$

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

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

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  $action[1, +] = shift(s_6)$ .

$I_2$  generates  $action[s_2, \$] = action[s_2, +] = action[s_2, )] = reduce(E \rightarrow T)$  since  $follow(E) = \{\$, +, )\}$  and  $action[2, *] = shift(s_7)$ .

etc.

## SLR(1) Grammars 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

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

The first item in  $I_2$  sets  $action[s_2, =]$  to *shift*. The second item in  $I_2$  sets  $action[s_2, =]$  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 VIII

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  $w = a_1a_2 \cdots a_n$ .

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

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

- 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  $T[i, i + 1]$ .
- $w$  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 ⇒ aj}
  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
```

## 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 $n$*  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.

## 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.

## Converting Inherited Attributes

Consider this fragment of our expression attribute grammar:

$$\begin{aligned} term &\rightarrow factor\ term' \\ &\quad term'.v1 := factor.v; term.v := term'.v; \\ term' &\rightarrow *term \\ &\quad term'.v := term'.v1 * term.v; \\ term' &\rightarrow \epsilon \\ &\quad term'.v := term'.v1; \\ factor &\rightarrow \mathbf{int} \\ &\quad factor.v := int.v; \end{aligned}$$

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

$$\begin{aligned} term &\rightarrow factor\ term' \\ &\quad term.val := term'.fn(factor.val); \\ term' &\rightarrow *term \\ &\quad term'.fn := (fn\ t => t * term.val); \\ term' &\rightarrow \epsilon \\ &\quad term'.v := fn\ t => t; \\ factor &\rightarrow \mathbf{int} \\ &\quad factor.val := int.val; \end{aligned}$$

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
```

```
%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.

## 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 (se 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).

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

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



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.

## 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

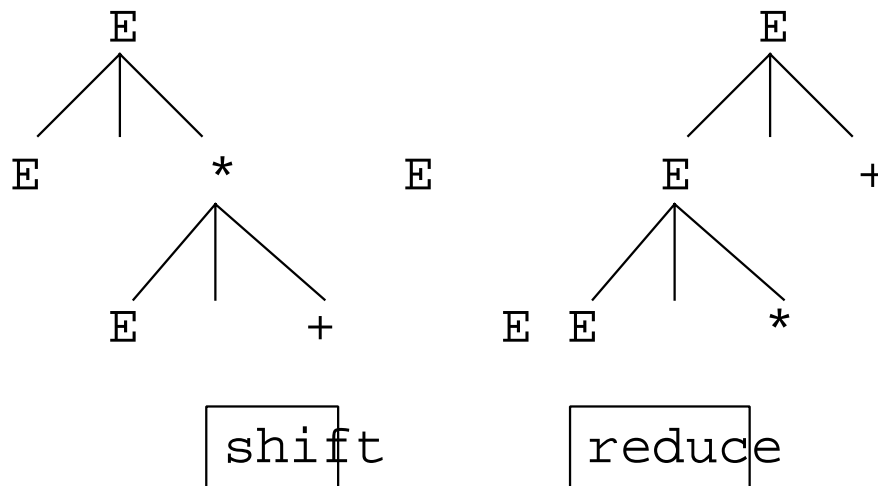
$$\begin{aligned} E &\rightarrow E * E. \quad (\textit{lookahead } +) \\ E &\rightarrow E. + E \end{aligned}$$

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

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

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:

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

*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")
```

## 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

## 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) ,
```

```

                                !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.

## 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

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";
```

## 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
- 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
- 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
```

## 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
```

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  $PRINT3 + 4 * -5$ .

# Programming Language Implementation IX

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.

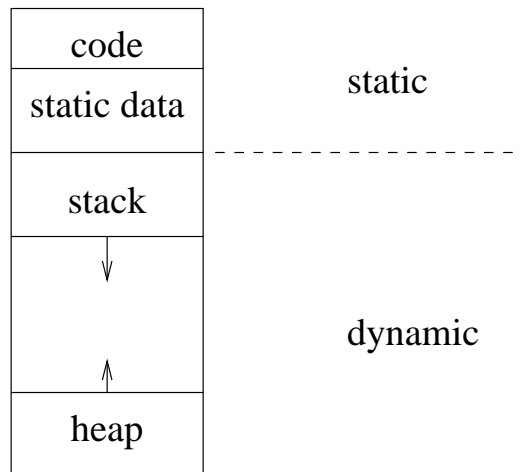
**FORTRAN:** The size of data structures is determined at compile time and sub-programs cannot be recursive.

**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.)

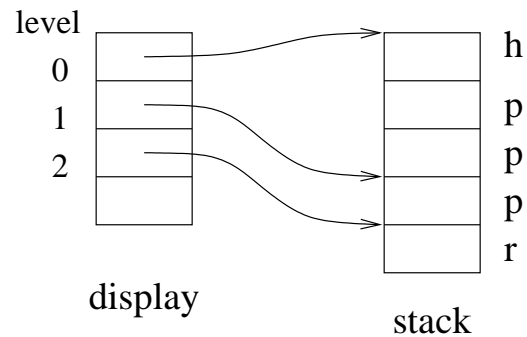
The **textual level** of a procedure (block) is the number of procedures surrounding it.

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 off set  $x$ .

**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.

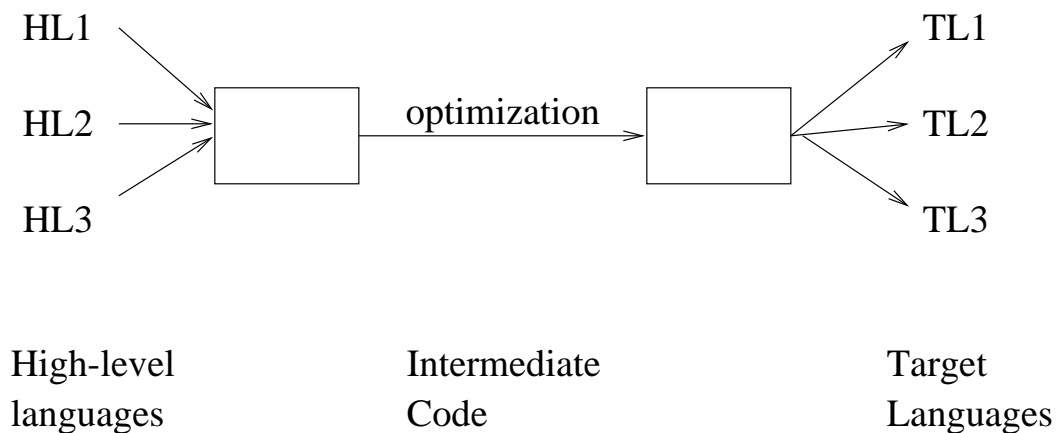
## Code Generation

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**

## 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.

## Intermediate Code

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

$$x := y \langle \text{op} \rangle z$$

Thus a complex source language expression like

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

will give rise to the statements

$$\begin{aligned} t1 &:= u * z \\ x &:= y + t1 \end{aligned}$$

Three address statements are similar to assembly code:

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

$$x := \langle \text{op} \rangle z$$

where  $\langle \text{op} \rangle$  is a unary operation.

- **Copy statements** of the form

$$x := z$$

- **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.

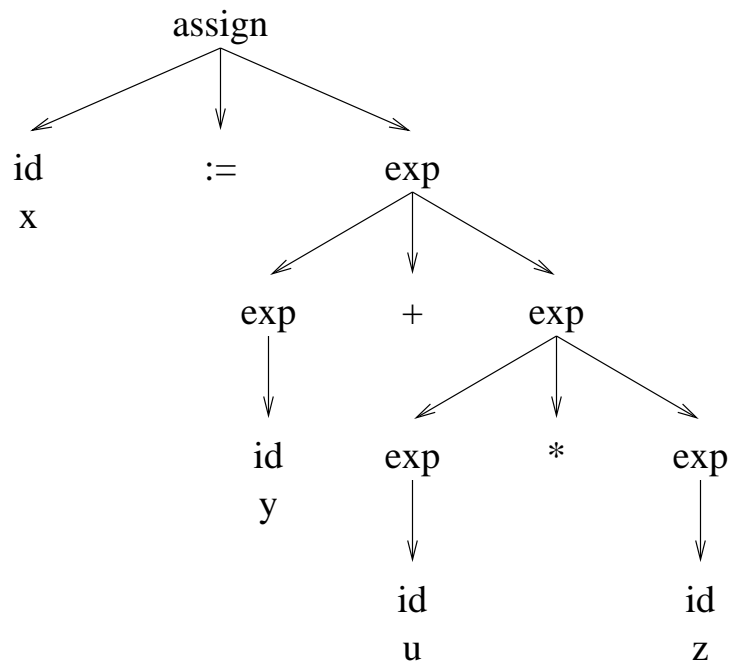
Production	Semantic Rules
$assgn \rightarrow \mathbf{id} := exp$	$assgn.code :=$ $exp.code @ gen(\mathbf{id}.place := exp.place)$
$exp_0 \rightarrow exp_1 + exp_2$	$exp_0.place := newtemp$ $exp_0.code :=$ $exp_1.code @ exp_2.code @$ $gen(exp_0.place := exp_1.place + exp_2.place)$
$exp_0 \rightarrow exp_1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

## 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!).