LR(1) Items

- New item form: \([A \rightarrow \alpha, \beta, a] \) a terminal (including $\)$
- \(a\) represents (1 token) lookahead
- Used only if \(\beta = \varepsilon\), i.e., we use lookahead to decide whether to reduce (or shift/error)
- \(a \in \text{Follow}(A)\)

**LR(1) Items Construction**

**Function closure(I):**

```plaintext
begin
  repeat
    for each item \([A \rightarrow \alpha. B \beta, a]\) in I,
    each production \(B \rightarrow \gamma\) in \(G\)',
    and each terminal \(b\) in \(\text{First}(\beta a)\)
    add \([B \rightarrow \gamma, b]\) to I (if not present)
  until no more items can be added to I
  procedure items('')
  begin
    C := \{\text{closure({S' -> .S, $})}\}
    repeat
      for each set of items I in C and each grammar symbol X such that
      \(\text{goto}(I, X)\) is not empty do
        add \(\text{goto}(I, X)\) to C (if not there)
      until no more items can be added to C
  end;

  function goto(I, X);
  begin
    let \(J\) be the set of items \([A \rightarrow \alpha, X. a, a]\) such that
    \([A \rightarrow \alpha, X. a] \) is in I
    let \(J\) to I (if not present)
    until no more items can be added to I
    end;
```
Example

- Consider same grammar again:
  - $S \rightarrow L = R$
  - $S \rightarrow R$
  - $L \rightarrow *R$
  - $L \rightarrow \text{id}$
  - $R \rightarrow L$

LALR Parsing

- Advantages
  - Smaller parsing tables than canonical LR
    - 100s versus 1,000s for typical language
  - Most PL constructs can be parsed with LALR
    - Algorithm of choice in practice
- Construction idea
  - Merge different items with same core, e.g.
    - $L \rightarrow \text{id}$, $\$ and $L \rightarrow \text{id}$, $+$
  - May reduce where an error should be reported
    - But reduction will lead to a parse error later
    - Merging does not produce new shift-reduce conflicts
      - Possible reduce-reduce conflicts; merge only if no conflicts

LALR Table Construction

- Simple but space / time consuming
- Build LR tables
- Merge item sets with identical cores
  - If conflict results: grammar not LALR
Efficient LALR Construction

- Ideas:
  - Represent only kernels for items set I, i.e., either
    $S' \Rightarrow S$ or item with ",, not leftmost
  - Can generate all items in I from kernel
  - Can generate parsing actions from kernel

- Details in the book, in case you are interested
  - won’t be on exam / prelim

Parsing with more Lookahead

- LR(k)
  - Closure: $A \Rightarrow \alpha\beta, \text{ a predict } B \Rightarrow \gamma, \text{ y where}
    y \in \text{First}_k(\beta)\text{x}$
  - Reduce: $A \Rightarrow \alpha, \text{ x reduce on lookahead } x$
  - Shift on lookahead $x$ for $A \Rightarrow \alpha\beta, \text{ y, where } a \text{ is a terminal and } x \in \text{First}_k(\alpha\beta)\text{y}$

- SLR(k)
  - Reduce: $A \Rightarrow \alpha, \text{ if lookahead } x \in \text{Follow}_k(A)$
  - Shift on lookahead $x$ for $A \Rightarrow \alpha\beta, \text{ a terminal and } x \in \text{First}_k(\alpha\beta\text{Follow}_k(A))$

- LALR(k): merge LR(k) machine states with same core

LR(k) in Practice

- LR(k), SLR(k) are not used in practice for k>1
  - Tables too large
  - Not necessary in practice since most grammars can be made LR(1) and even LALR(1)

- Some parser generators for LALR(2)
  - Useful if too lazy to rewrite the grammar
Parsing with Ambiguous Grammars

- Use non-grammatical means to resolve conflict
- Want to have just ONE parse tree!
- Useful because grammar can be
  - "more natural"
  - Smaller
- Conflict resolution:
  - Precedence
  - Associativity
  - Shift versus reduce

Precedence Example

\[ E \rightarrow E + E \mid E \cdot E \mid (E) \mid id \]

versus

\[ E \rightarrow E + T \mid T \]

\[ T \rightarrow T \cdot F \mid F \]

\[ F \rightarrow (E) \mid id \]

Parse for id + id * id
Parser Generators

- Yacc, bison, LLGen, LRGen etc
- Specify grammar
  - Produce a table-drive parser
  - Typically can use precedence & associativity rules and allow shift-reduce conflicts
- Project uses Cup a parser generator for Java

Syntax-directed Translation & Type checking

Background Reading

- Chapter 5: read 5.1 - 5.4
  - Rest of chapter is optional -- won't be tested in exam
- Chapter 6: complete
Syntax-directed Definition

- Add attribute(s) to grammar symbols
  - E.g., type, value, memory location
- Semantic rules
  - Compute attributes from rhs of production (synthesized attribute)
  - Pass attribute from lhs to rhs (inherited attribute)
- Intrinsic attributes (assigned to leaves)
  - Sometimes also called synthesized (but from "external info")
- Decorating/annotating the parse tree = computing attributes for all parse tree nodes

Syntax-directed definitions and Grammars

- Attribute Grammar = syntax-directed definition w/o side-effects
- S-attributed definition
  - A syntax-directed definition where all attributes are synthesized

Example

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>L -&gt; E n</td>
<td>print(E.val)</td>
</tr>
<tr>
<td>L-&gt;E₁ + T</td>
<td>E.val = E₁.val + T.val</td>
</tr>
<tr>
<td>E-&gt;T</td>
<td>E.val = T.val</td>
</tr>
<tr>
<td>T-&gt;T₁*F</td>
<td>T.val = T₁.val * F.val</td>
</tr>
<tr>
<td>T-&gt;F</td>
<td>T.val = F.val</td>
</tr>
<tr>
<td>F-&gt;(E)</td>
<td>F.val = E.val</td>
</tr>
<tr>
<td>F-&gt;digit</td>
<td>F.val = digit.lexval</td>
</tr>
</tbody>
</table>
Inherited Attributes

- Value from a parent or sibling in parse tree
- Used to express dependences of PL constructs
- E.g. types, lvalue vs. rvalue

Example: Inherited Attributes

Production | Semantic Rules
---|---
D -> T L | L.in := T.type
T -> int | T.type := integer
T -> real | T.type := real
L -> L1, id | L1.in := L.in
              | addtype(id.entry, L.in)
L -> id | addtype(id.entry, L.in)

Dependency Graphs

- Semantic rules produce dependencies, e.g., b := f(c1, ..., cn) can compute b only after computing all ci
- Graphical representation
  - Directed graph with an edge from ci to b
- Evaluation order
  - Topological sort
  - Rule-based (fixed at compile time)
  - Oblivious (does not work for all syntax-directed definitions)
L-attributed Definitions

- A -> X_1 X_2 ... X_n
  - Attribute of X_i depends only on inherited attributes of A and on X_1 ... X_{i-1}
  - Can be evaluated by a left-to-right (recursive) traversal

Semantic Analysis

- Type checks
  - Statically or dynamically
- Control-flow checks
  - E.g., no goto into the middle of a (different procedure)
- Uniqueness checks
  - E.g., redefinition of variables
- Name (matching) checks
  - Procedure name (e.g., Modula 2)

Type checking

- Check
  - That operands have expected types
  - Expressions are well-typed
- What is a type?
  - "A Collection of computational entities that share some common property" (Mitchell)
  - Determines the range of values and set of operations that are defined for values of the type
Type Systems

- Rules for assigning types to programming language constructs
- "A type system is a syntactic method for enforcing levels of abstraction in programs." (Benjamin C. Pierce)

Why Types?

- Program organization and documentation
- Consistent interpretation of bits in memory (avoid unintended semantics)
- Optimization
  - Compiler can exploit type information for better performance, example:
  - Type \( r = \) record name: array[1..256] of char;
  - year: integer;
  - end;
  - Can generate efficient code to access record fields:
  - address(s.year) = base_address(s) + 256 * sizeof(char)

Basic & Composite Types

- Basic types
  - Depend on PL, usually integers, reals, characters, booleans etc.
- Type constructors
  - Used to build new types from existing ones
  - Common constructors:
    - Array types
    - Product types (unnamed tuples)
- Record types
  - Named tuples
  - E.g. type row = record
    - address: integer;
    - lexeme: array[1..15] of char;
  - end;
  - Record types (named tuples)
  - Product types (unnamed tuples)
  - Pointers
Type Expression Definition
- A type expression is one of
  - Basic type
  - Type name
  - Type constructor
  - Type variable

Type System revisited
- Set of rules to assign types
  - Implemented by a type checker
- Type checking can be done
  - Statically (= at compile time)
    - Modula 2, C, Java -> statically typed language
    - In practice have to do some checks at run-time as well,
      e.g. array bounds checking a[i] is i within bounds?
  - Dynamically (when the program is running)
    - Lisp -> dynamically typed language

Strong vs. Weak
- Strongly typed language
  - A program that passes the type checker
    will not have a run-time (type) error
  - Example: Java
- Weakly typed language
  - Can have type errors
    - C, C++
Declared vs. Inferred Types

- Most programming languages require the programmer to declare types
  - E.g., int x,y,z; int foo(int x)...
- Some language infer all types based on inference rules
  - Eg. ML (statically typed but don’t have to declare types)

Example: TC Expressions

\[
\begin{align*}
E &\rightarrow \text{literal} \{ E.\text{type} := \text{char} \} \\
E &\rightarrow \text{num} \{ E.\text{type} := \text{integer} \} \\
E &\rightarrow \text{id} \{ E.\text{type} := \text{lookup(id.entry)} \} \\
E &\rightarrow E_1 \ [E_2] \{ E.\text{type} := \text{if } E_2.\text{type} = \text{integer} \text{ and } E_1.\text{type} = \text{array(s,t)} \text{ then t else type_error} \} \\
E &\rightarrow E_1 \ ^\{ \text{if } E_1.\text{type} = \text{pointer(t) then t else type_error} \}
\end{align*}
\]

Type Compatibility

- What are the types that can legally be used in a particular context?
  - E.g. 2.5 + x, what type should x be?
- Type conversion ("cast")
  - 2.5 + (float) x
  - Widening (int to float) versus narrowing (float to int)
- Type coercion
  - Implicit conversion dictated by language rules
Structural Equivalence
- Two types are compatible if either of the same basic type or built by same constructor applied to structurally equivalent types

Name Equivalence
- Named types
  - Types equivalent only when they have the same name
  - Unnamed types get fresh names associated with their declaration
    - type A = array[1..10] of integer;
    - type B = array[1..10] of integer;
    - var x,y: A;
    - var u,v: B;
    - var w: array of [1..10] of integer;
    - Only x ≡ y and u ≡ v

Comparison
- Structural equivalence
  - More flexible
  - More expensive to check
- Name equivalence
  - Efficiently checkable
  - More restrictive
Polymorphism

- Parametric polymorphism
  - Function can be applied to any arguments whose type match a type expression with type variables
- Ad hoc polymorphism
  - (aka overloading) two or more implementations with different types referred to by same name (e.g., + can be integer and floating point addition)
- Subtype polymorphism
  - Subtype relation between types allows an expression to have many possible types
  - Common in oo-languages, e.g. COOL

Parametric Polymorphism

- Characteristic: type variables
- Often found in functional languages, e.g. ML: sort : ('a * 'a -> bool) * 'a list -> 'a list
- Function may have infinitely many types
- Implicit or explicit
  - ML implicit: programmer does not have to declare types
  - Type inference computes necessary types
  - Explicit: program text contains type variables, e.g., C++ templates

C++ Templates

Template <T>
void swap (T& x, T& y) {
    T tmp = x; x = y; y = tmp;
}
Implementing Parametric Polymorphism

- C++ (explicit)
  - Templates are instantiated at link time (why?)
- Different copies of polymorphic function (e.g., swap) are created
  - Why?
- One of the major reasons for the complexity of C++ linkers

Implementing Parametric Polymorphism (2)

- ML (implicit)
- No need for multiple function copies
  - Uniform data representation for all types: "boxed"
  - Advantage:
    - No code duplication
  - Disadvantage
    - Performance degradation - can be mitigated by "unboxing optimization"