CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
symbol tables

One table per scope

Solid interface functions (constructors, accessors and mutators)

Good encapsulation and information hiding

Flexible design
struct SymbolTable;
struct SymbolTableList;
struct SymbolTableEntry;

/* Every symbol table entry must denote either a TYPE, a FUNCTION, or a VARIABLE. */

The type EntryCategory is used to express the kind of symbol table entry:

    TYPE is used for entries that denote types
    FUNCTION is used for entries that denote functions
    VARIABLE is used for entries that denote variables

*/
enum EntryCategory { TYPE, FUNCTION, VARIABLE };

/* Every type belongs to one of the following categories: */

    PRIMITIVE is used for primitive types (such as integer, real, character, Boolean)

    PRODUCT is used for Cartesian products of types (i.e. structs/records)

    SUM is used for union (or sum) types; alpha does not currently support this category of type.

    MAPPING is used for mapping types: function types and array types

    UNDEFINED is used for expressions whose type is ill-defined

*/
enum TypeCategory { UNDEFINED, MAPPING, PRIMITIVE, PRODUCT, SUM };
Constructors
These functions build new values of the type indicated by the return type specification.

/* Build and return a pointer to a new SymbolTable. Every symbol table has a unique parent, except the top-level symbol table. The top-level symbol table is created by the call:

    newSymbolTable(NULL)

*/

struct SymbolTable* newSymbolTable(struct SymbolTable* parent);

/* Build and return a pointer to a new SymbolTableList. The SymbolTableList has one member, table.

*/

struct SymbolTableList* newSymbolTableList(struct SymbolTable* table);

/* Build and return a pointer to a new SymbolTableEntry, of the indicated category.

*/

struct SymbolTableEntry* newSymbolTableEntry(enum EntryCategory category);
Mutators

```c
void addEntryToSymbolTable(struct SymbolTable* table, struct SymbolTableEntry* entry);
void addChildToSymbolTable(struct SymbolTable* parent, struct SymbolTable* child);
```
/******************************************************************************
Accessors
*******************************************************************************/

struct SymbolTable* getSymbolTable(void);
struct SymbolTable* getParent(struct SymbolTable* table);
struct SymbolTableList* getChildren(struct SymbolTable* table);
struct SymbolTableList* getRestOfChildren(struct SymbolTableList* list);
struct SymbolTable* getFirstOfChildren(struct SymbolTableList* list);
struct SymbolTableEntry* getEntryInSymbolTable(struct SymbolTable* table, char* name, bool ancestorSearch);
char * getName(struct SymbolTableEntry* entry);
enum EntryCategory getEntryCategory(struct SymbolTableEntry* entry);
enum TypeCategory getTypeCategory(struct SymbolTableEntry* entry);
struct SymbolTableEntry* getType(struct SymbolTableEntry* entry);
bool hasInit(struct SymbolTableEntry* entry);
int_least32_t makeSymbolTableID(int lineNumber, int colNumber);
struct SymbolTable* getSymbolTable(struct SymbolTableEntry* entry);
Phases of a compiler

Figure 1.7, page 5 of text

Syntactic structure

- Character stream
- Lexical Analyzer
- Token stream
- Syntax Analyzer
- Syntax tree
- Semantic Analyzer
- Syntax tree
- Intermediate Code Generator
- Intermediate representation
- Machine-Independent Code Optimizer
- Intermediate representation
- Code Generator
- Target-machine code
- Machine-Dependent Code Optimizer
- Target-machine code
Example 4.51 [p. 260]

Grammar from example 4.48:

\[
S \rightarrow L = R | R \\
L \rightarrow *R | id \\
R \rightarrow L
\]

\[
I_0: \quad S' \rightarrow \cdot S \\
S \rightarrow \cdot L \cdot R \\
S \rightarrow \cdot R \\
L \rightarrow \cdot * R \\
L \rightarrow \cdot id \\
R \rightarrow \cdot L
\]

\[
I_5: \quad L \rightarrow \cdot id. \\
I_6: \quad S \rightarrow L = \cdot R \\
R \rightarrow \cdot L \\
L \rightarrow \cdot * R \\
L \rightarrow \cdot id \\
I_7: \quad L \rightarrow \cdot id \\
I_8: \quad R \rightarrow \cdot L \\
I_9: \quad S \rightarrow L = \cdot R
\]

Figure 4.39: Canonical LR(0) collection for grammar (4.49)
Example 4.51 [p. 260]

Grammar from example 4.48:

\[
\begin{align*}
S & \rightarrow L = R \mid R \\
L & \rightarrow *R \mid id \\
R & \rightarrow L
\end{align*}
\]

"[This grammar] is not ambiguous. This shift/reduce conflict arises [because] SLR parser construction method [does not] remember enough left context..." [p. 255]
Viable prefix

"Why can LR(0) automata be used to make shift-reduce decisions? The LR(0) automaton for a grammar characterizes the strings of grammar symbols that can appear on the stack... The stack contents must be a prefix of a right-sentential form. If the stack holds $\alpha$ and the rest of the input is $x$, then a sequence of reductions will take $\alpha x$ to $S$. In terms of derivations, $S \Rightarrow_{rm*} \alpha x$." [p. 256]
Viable prefix

"Not all prefixes of right-sentential forms can appear on the stack...since the parser must not shift past the handle." [p. 256]

\[ E \Rightarrow_{rm^*} F \ast \ id \Rightarrow_{rm} \ ( E ) \ast \ id \]
Viable prefix

"Not all prefixes of right-sentential forms can appear on the stack...since the parser must not shift past the handle." [p. 256]

\[ E \Rightarrow_{rm*} F \ast id \Rightarrow_{rm} (E) \ast id \]

(\( E \)) is a handle of \( F \rightarrow (E) \)
Viable prefix
(paraer configurations shown)

($, 'id') * id $

$ '(' id ')' * id $

$ '(' id , ')' * id $

$ '(' F , ')' * id $

$ '(' T , ')' * id $

$ '(' E , ')' * id $

$ '(' E ')' , * id $

$ F , * id $

$ T , * id $

$ T * , id $

etc.

Cannot shift '*' here, because

'( E )'
is a handle.
Viable prefix

"The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes." [p. 256]
Viable prefix

\[
(\$, (\text{id })^\ast \text{id})
\]
\[
(\$, (\text{id })^\ast \text{id})
\]
\[
(\$, (\text{id })^\ast \text{id})
\]
\[
(\$, (\text{F })^\ast \text{id})
\]
\[
(\$, (\text{T })^\ast \text{id})
\]
\[
(\$, (\text{E })^\ast \text{id})
\]
\[
(\$, (\text{E }')^\ast \text{id})
\]
\[
(\$, \text{F }^\ast \text{id})
\]
\[
(\$, \text{T }^\ast \text{id})
\]
\[
(\$, \text{T }^\ast , \text{id})
\]

etc.

Cannot shift \('*'\) here, because
\((\' \text{E }')\)

is a handle.

Therefore
\((\' \text{E }')\) *

is not a viable prefix.
LR(1) items

"...in the SLR method, state I calls for reduction by $A \rightarrow \alpha$ if the set of items $I_i$ contains item $[A \rightarrow \alpha \bullet]$ and input symbol $a$ is in FOLLOW($A$)." [p. 260]
"In some situations, however, when state I appears on top of the stack the viable prefix $\beta \alpha$ on the stack is such that $\beta A$ cannot be followed by $a$ in any right-sentential form." [p. 260]
Example 4.51 [p. 260]

Grammar from example 4.48:
\[ S \rightarrow L = R \mid R \
L \rightarrow *R \mid id \\
R \rightarrow L \]

State I₂ from figure 4.39
\[ S \rightarrow L \cdot = R \\
R \rightarrow L \cdot \]

"Consider the set of items I₂. The first item in this set makes ACTION[2,=] be 'shift 6'. Since FOLLOW(R) contains = [...] the second item sets ACTION[2,=] to reduce R \rightarrow L." [p. 255]

"...the SLR parser calls for reduction by R \rightarrow L in state 2 with = as the next input (the shift action is also called for ...). However, there is no right-sentential form of the grammar ... that begins R = ... . Thus state 2, which is the state corresponding to viable prefix L only, should not really call for reduction of that L to R." [p. 260]

See section 4.7.5 (p. 270) for more discussion of this example.
"By splitting states when necessary, we can arrange to have each state ... indicate exactly which input symbols can follow a handle $\alpha$ for which there is a possible reduction to A." [p. 260]

"The general form of an item becomes

$[ A \rightarrow \alpha \cdot \beta, a ]$

where $A \rightarrow \alpha \beta$ is a production and $a$ is a terminal or \ldots $\$.$" [p. 260]
"The lookahead has no effect in an item of the form \([ A \rightarrow \alpha \bullet \beta, a]\), where \(\beta\) is not \(\epsilon\), but an item of the form \([ A \rightarrow \alpha \bullet, a]\) calls for reduction by \(A \rightarrow \alpha\) only if the next input symbol is \(a\). [...] The set of such \(a\)'s will always be a subset of \(\text{FOLLOW}(A)\), but it could be a proper subset ..." [p. 260]
LALR (lookahead LR)

"SLR and LALR tables ... always have the same number of states." [p. 266]

Idea: merge sets of LR(1) items with the same core.

Cannot introduce Shift/Reduce conflicts, may introduce Reduce/Reduce conflicts.

Bison and YACC produce LALR parsers.
Phases of a compiler

Semantic analysis

Figure 1.6, page 5 of text
Semantics

• “Semantics” has to do with the meaning of a program.

• We will consider two types of semantics:
  
  – Static semantics: semantics which can be enforced at compile-time.
  
  – Dynamic semantics: semantics which express the run-time meaning of programs.
Static semantics

- Semantic checking which can be done at compile-time

- Type-compatibility is a prime example
  - `int` can be assigned to `double` (type coercion)
  - `double` cannot be assigned to `int` without explicit type cast

- Type-compatibility can be captured in grammar, but only at expense of larger, more complex grammar
Ex: adding type rules in grammar

• Must introduce new non-terminals which encode types:
• Instead of a generic grammar rule for assignment:
  - \( <\text{stmt}> \rightarrow <\text{var}> \=' <\text{expr}> \ ';\)
• we need multiple rules:
  - \( <\text{stmt}> \rightarrow <\text{doubleVar}> \=' <\text{intExpr}> \mid <\text{doubleExpr}> \ ';\)
  - \( <\text{stmt}> \rightarrow <\text{intVar}> \=' <\text{intExpr}> \ ';\)
• Of course, such rules need to handle all the relevant type possibilities (e.g. byte, char, short, int, long, float and double).
Alternative: attribute grammars

• Attribute grammars provide a neater way of encoding such information.
• Each syntactic rule of the grammar can be decorated with:
  – a set of semantic rules/functions
  – a set of semantic predicates
Attributes

- We can associate with each symbol X of the grammar a set of attributes A(X). Attributes are partitioned into:

  synthesized attributes S(X) – pass info up parse tree

  inherited attributes I(X) – pass info down parse tree
Semantic rules/functions

- We can associate with each rule R of the grammar a set of semantic functions.

- For rule \( X_0 \rightarrow X_1 \ X_2 \ \ldots \ \ X_n \)
  - synthesized attribute of LHS:
    \[ S(X_0) = f(A(X_1), \ A(X_2), \ \ldots, \ A(X_n)) \]
  - inherited attribute of RHS member:
    for \( 1 \leq j \leq n \), \[ I(X_j) = f(A(X_0), \ldots, A(X_{j-1})) \]
    (note that dependence is on siblings to left only)
Predicates

- We can associate with each rule $R$ of the grammar a set of semantic predicates.

- Boolean expression involving the attributes and a set of attribute values

- If `true`, node is ok

- If `false`, node violates a semantic rule