An Object-Oriented Specification for Compiler

Pei-Chi Wu Feng-Jian Wang
Institute of Computer Science and Information Engineering
National Chiao Tung University
1001 Ta Hsueh Road, Hsinchu, Taiwan, R.O.C.
FJWANG@TWNCTU01.BITNET

ABSTRACT

In this paper, we present an object-oriented approach to compiler specification. Our method treats grammar symbols as templates which instantiate nodes of parse tree for object-oriented semantic analysis. To have better classification and inheritance for semantic description, it uses *restricted* CFG to specify grammar rules. The semantic specification is done based on a class hierarchy generated from the restricted CFG. Besides, that a path expression is booted to describe the possible receivers of a message releases the restriction on the specification of message (value) propagations in attribute grammar methods.

Keywords: object-oriented, compiler, specification, context-free grammar, attribute grammar, classification, inheritance, Smalltalk-80.

1. INTRODUCTION

Although Knuth showed that even the *purely synthesized AG* has the same power as any other *AG* does [4], researches on the extension of attribute grammar to simplify the specification of attribution rules have never stopped [6, 8, 9]. However, there was little attention based on object-oriented programming [7, 10].

Consider the productions in a CFG, if a nonterminal symbol X has only the following production form $X \rightarrow X_1 \mid X_2 \mid \ldots \mid X_k$, X and $X_1\ X_2\ \ldots\ X_k$ may be treated as a class hierarchy, where class X is the parent class of classes $X_1\ X_2\ \ldots\ X_k$. Therefore, the attributes and attribution rules specified to be associated with X can be inherited by $X_1\ X_2\ \ldots\ X_k$, if the nodes of parse tree are treated as objects of these classes. The details of applying classification and inheritance on a CFG for compiling are shown in [11]. The work in [5] applies similar concepts to code a compiler.

The value propagations along parse tree follow either implicit orders, based on conventional attribute grammars, specified by attributes assignments or global data, or the pre-specified order which is defined based on graph. In the latter case, Demers *et al.*, proposed a method which specifies the propagation directions according to the tree abstraction of parse tree [2]. It allows the attributes values of a node in the parse tree be sent to their parents, preorder predecessors, inorder successors, ..., *etc.*, directly. However, none of these methods allow the propagation along an arbitrary path which is explicitly defined during compiler specification.

Perhaps the major difficulty to specify the sending of a value through an arbitrary path is that the parse tree can not be decided at specification. The language needed to define the paths associated with a CFG seems so complicated that there is no good result of research working on it yet. Our experience shows that a restricted CFG and the corresponding class hierarchy can ease the specification of paths. Here, we present a method to describe path informations based on the class hierarchy and *structured* parse tree [11]. The message, defined with a path, is thus sent to the destination node accordingly. Besides, perhaps the most interesting thing is that the specification of message paths in our method is easy.

2. THE MODEL

Let $G = (T, N, P, Z)$ be a CFG, where T is a *terminal* symbol set, N a *nonterminal* symbol set, P a *production* set, and Z a *start* symbol. A production in a CFG is a *singleton* if its right-hand side is a single symbol. It is a *skeleton* in other cases except an empty production. A *production set* of nonterminal X,

denoted as $P(X)$, is a set of all the productions of the form $X \rightarrow \alpha$, where α is a string of symbols from $(T \cup N)^*$.

Definition 2.1: A nonterminal symbol X in a CFG is:

1. a *classification* symbol if $\forall p \in P(X)$ is singleton,
2. a *structure* symbol if $|P(X)|=1$ and $p \in P(X)$ is skeleton, or
3. a *compound* symbol if $|P(X)|>1$ and $\exists p \in P(X)$ is skeleton.

As discussed in [11], classification and structure symbols can be analogized as the classes of a class hierarchy with inheritance so that a classification symbol and its righthand side represents a specialization/generalization in a (object-oriented) class hierarchy. A structure symbol and its righthand side can be considered as a branch of parse tree. Compound symbols are jammed with too much information that they may be deleted from the grammar to simplify the structure of the class hierarchy. This restriction may lengthen grammar specification, but not the expression power of the grammar.

Definition 2.2: Let $G = (T, N, P, Z)$ be a CFG. G is a *restricted* CFG if it contains no compound symbol.

Ambiguous grammars may make the compiler specification shorter and more natural for a language. But, it does little help in the discussion of our model. Neither useless symbols nor ε-productions do. Therefore, our scope is in restricted CFG without these properties.

A class hierarchy can be generated based on a restricted CFG, so that the nodes of parse tree are treated as the instances of its classes [11]. Node objects pass information to each other for semantic analysis, code generation/optimization, ..., etc. Node objects are instantiated during parsing process, and their addresses are not known at specification time. However, the distance between two node objects can be specified based on tree abstraction. In order to simplify the specification, we define a regular expression, *path expression*, to describe the paths to receiver nodes of a message from its sender. During semantic analysis, a path expression and sender node of a message is interpreted as its receiver nodes so that the message can be delivered.

Definition 2.3: Let r and s be path expressions denoting the languages R and S, respectively.

1. $\varnothing$ is a path expression and denotes the empty set.
2. ε is a path expression and denotes the set $\{\varepsilon\}$.
3. A *symbol index* (c, i) is a path expression and denotes the set $\{(c, i)\}$, where c is a symbol, i is a *direction* index,
4. rs is a path expression, which denotes the set RS, i.e. $\{ a_1 a_2 \mid a_1 \in R, a_2 \in S\}$, where $a_1 a_2$ represents the concatenation of strings a_1 and a_2.
5. $r+s$ is a path expression, which denotes the set $R \cup S$, where '+' called *alternative* operator.
6. r^* is a path expression, which denotes the closure of R, R^*, i.e. set $\{\varepsilon\} \cup R \cup RR \cup ... \cup RR...R \cup ...$, where '*' called *repetitive* operator.

The *alphabet set* of path expression is a set of symbol index (c, i). The direction index is an integer, which represents traveling direction on a parse tree. The positive direction index i means traveling to the ith child which labeled c. The negative index $-i$ means traveling to parent labeled c and ensures that the node itself is the ith child of its parent.

Let's assume that each parent-child relation in the tree is associated with two directed arcs, of which each is labeled a symbol index. In a graph, a node v is *reachable* from u if there is a path from u to v. Since a node in parse tree is an instance of a symbol, a path in parse tree can also be deemed as an instance of a sequence of symbol indexes. A node v is a *reachable end* from a node u through s, a sequence of symbol indexes, if the path from u to v is s's instance. A node v is a *receiving node* from a sender node u through a path expression e, if there is a sequence of symbol indexes s in the denoting language of e, s.t. v is a reachable end from u through s. The receiving nodes from a sender node with a path expression will be treated as the *receivers*.

Besides, some short-cut notations and frequent expressions are introduced to simplify the

specification of path expression. Two characters '+' and '-' are used in direction index to represent positive and negative direction index, respectively. A symbol *c* in *G* is equivalent to notation (*c*, +), which means any child labeled *c*. The special symbol *any* means any symbol. The *self* symbol, equal to an empty string ε, means the node itself. The '^' character prefixed a symbol *c* in *G* means any symbol but *c*.

Figure 2.2 shows a parse tree of a CFG in figure 2.1. For example, a path expression "(<expr>,3) (<id>,+)" associated with node <assign>$_1$ is a language $\mathcal{L}$={ (<expr>,3) (<id>,1), (<expr>,3) (<id>,3)}. Since, from the <assign>$_1$ node, there exist a path through arcs labeled (<expr>,3) and (<id>,1) to node <id>$_2$, and another path through arcs labeled (<expr>,3) and (<id>,3) to <id>$_3$, respectively, <id>$_2$ and <id>$_3$ are both receiving nodes from <assign>$_1$ through "(<expr>,3) (<id>,+)". In other words, the expression "(<expr>,3) (<id>,+)" associated with node <assign>$_1$ represents the receiver node set {<id>$_2$, <id>$_3$}.

```
<assign> ::= <id> := <expr>
<expr> ::= <id> + <id>
```

Figure 2.1: A CFG.

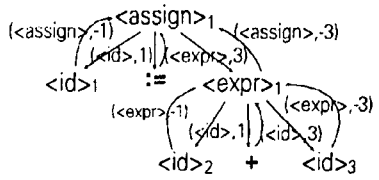

Figure 2.2: Symbol indexes in a parse tree.

Definition 2.4: A *site definition* is a definition *r=e*, where *r* is a symbol called *pseudo site*, and *e* is a path expression.

The semantics specification in our model contains two parts: *site definitions* and *service descriptions*. Pseudo sites are defined to represent path expressions. They are used as receivers in the service descriptions. The message sent to a pseudo site is applied to each node object in the set, which is similar to *Set* in Smalltalk-80. This message can be thought as an implicit *collect* message, which is sent to each element of the receiver node set. Its return is like a *Bag* in Smalltalk-80. Messages sent to an empty node set will return an empty bag. For example, pseudo site IdNodes in class <assign> of above example represents path expression "(<expr>,3) (<id>,+)". It is thought as a set of receivers corresponding to the real paths. A message *idName* sent to site IdNodes is interpreted as "{<id>$_2$, <id>$_3$} *collect*: [:n | n *idName*]".

Sometimes, a pseudo site is considered as a single node instead of a set, if the path expression defined is always bound to a set of single element. The implicit propagation directions, such as *parent* and *child*, in attribute grammar, are always bound to single element, if the symbol is not start or terminal symbol. The pseudo site AssignNode defined as "(^<assign>,-)* (<assign>,-)" on class <id> is a language { (<assign>,-), (<expr>,-) (<assign>,-), (<id>,-) (<assign>,-), ...}. It is always bound to the <assign> instance (node) which is an ancestor of the <id> instance (node). In these situations, it is convenient to use the notation of single node instead of set.

The semantic definition of a symbol is the description of services (methods) provided by it. The notations in the description are similar to those in Smalltalk-80. To simplify specification, a node object is assumed to have only those attributes whose value can not be changed after ready. These attributes can be specified implicitly, and thus Smalltalk's instance variable is not used. Besides, not all services send message(s) to other nodes. For example, a service may return integer zero directly. A service which returns the lexical information of a terminal node has its definition done in lexical part (as in Lex).

Definition 2.5: An *object-oriented compiler specification* is a 3-tuple, *OOCS* = (*G*, $\mathcal{S}$, $\mathcal{E}$).

1. *G* = (*T*, *N*, *P*, *Z*) is a *restricted* CFG.
2. $\mathcal{S} = \bigcup_{X \in T \cup N} \mathcal{S}(X)$ is a finite set of service descriptions.

3. $\mathcal{E} = \bigcup_{X \in T \cup N} \mathcal{E}(X)$ is a finite set of site definitions (path expressions).

Here, $\mathcal{S}(X)$ and $\mathcal{E}(X)$ are a set of service descriptions and a set of site definitions associated with each symbol X in G, respectively.

3. AN OBJECT-ORIENTED SPECIFICATION METHOD

In this section, we present a specification method for compilers based on our model. The method contains four major steps:

1. Write a *restricted* CFG for the target language.
2. Construct the *exclusive* class hierarchy for the restricted CFG.
3. Define the services provided by each symbol.
4. Define each pseudo site by a path expression.

The flow chart is shown in figure 3.1.

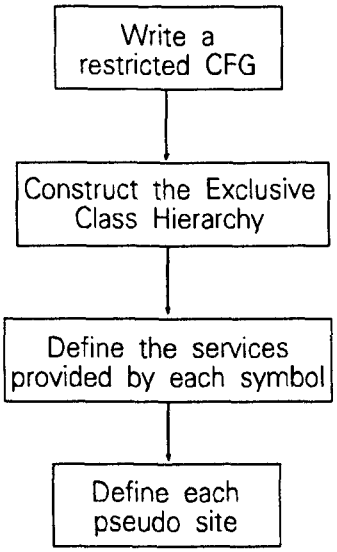

Figure 3.1: The specification flow chart.

3.1 The Specification Details

1. Write a *restricted* CFG for the target language.
   (a) Write a CFG for the target language.
   (b) Rewrite the production rules of compound symbols in the CFG to make an equivalent restricted CFG. For each compound symbol, modify its production rules by replacing their right-hand sides with new structure symbols. Make a production rule with each new structure symbol (left-hand side) and the corresponding right-hand side (see also [11]). Besides, name the introduced symbols carefully so that they can express the specializations of the compound symbol.
2. Construct the *exclusive* class hierarchy (ECH) for the restricted CFG.
   (a) Each singleton production $X \rightarrow Y$ claims that Y *exclusively* inherits from X, i.e. an object y of Y has the properties defined in X, the service descriptions and site definitions declared in X, if y is generated by $X \overset{*}{\underset{G}{\Rightarrow}} Y$.
   (b) Each skeleton production $X_0 \rightarrow X_1 X_2 \ldots X_k$, $k \geq 1$, describes that X_0 contains k components, X_1, X_2, ..., and X_k. An object x_i of X_i, $1 \leq i \leq k$, has the properties of *null*($\perp$) and X_i, if x_i is generated by $X_0 \overset{*}{\underset{G}{\Rightarrow}} X_1 X_2 \ldots X_i \ldots X_k$.
3. Define the services provided by each symbol.
   (a) Identify the services and their message protocols provided by each symbol; leave the symbol alone if nothing can be served.
   (b) Organize the services through generalization, i.e. moving common services to superclass(es)

in ECH.

(c) Specify each service with Smalltalk language, where message receivers are represented by pseudo sites.

4. Define each pseudo site by a path expression.
   (a) Sketch the propagation direction of the path. There are four kinds of paths: to itself, to ancestor(s), to offspring(s), or to arbitrary node(s) (offspring(s) of some ancestors).
   (b) Sketch some representative parse trees related to the pseudo site, and outline the shortest propagation paths in these parse trees.
   (c) Write down the expression according to these paths. Use the repetition (*) for repeated symbols, the alternative (+) for multiple choices, and the concatenation for sequence.

3.2 An Example

This subsection shows a specification example for the language ABL which contains assignment statements and block structures. The ABL compiler is expected to be capable of detecting naming errors or warnings, such as *un-declared*, *re-declared*, and *un-referenced* variables.

3.2.1 Restricted CFG

The first step is to write a CFG for ABL language. Figure 3.2 shows a sample which contains three compound symbols: <decl list>, <stmt list>, and <expr>. To make an equivalent restricted CFG, the structure symbols <stmt list branch>, <decl list branch>, and <add expr> are introduced. The production "<decl list> ::= <decl> <decl list>" is then replaced by "<decl list> ::= <decl list branch>" and "<decl list branch> ::= <decl> <decl list>". Similar work is treated on <stmt list> and <expr>. Besides, these structure symbols are named to express the specializations of the compound symbols. For example, <add expr> means additive expression, and <stmt list branch> means the branch in a statement list. The resulting restricted CFG is shown in figure 3.3.

```
<program> ::= PROGRAM ID <block>
<decl list> ::= <decl> | <decl> <decl list>
<stmt list> ::= <stmt> | <stmt> <stmt list>
<decl> ::= ID : TYPE
<stmt> ::= <assign> | <block>
<assign> ::= <id> := <expr>
<block> ::= BEGIN <decl list> <stmt list> END
<expr> ::= <expr> + <expr> | <id>
<id> ::= ID
```

Figure 3.2: The CFG of ABL language.

```
(1)  <program> ::= PROGRAM ID <block>
(2)  <decl list> ::= <decl> | <decl list branch>
(3)  <decl list branch> ::= <decl> <decl list>
(4)  <stmt list> ::= <stmt> | <stmt list branch>
(5)  <stmt list branch> ::= <stmt> <stmt list>
(6)  <decl> ::= ID : TYPE
(7)  <stmt> ::= <assign> | <block>
(8)  <assign> ::= <id> := <expr>
(9)  <block> ::= BEGIN <decl list> <stmt list> END
(10) <expr> ::= <add expr> | <id>
(11) <add expr> ::= <expr> + <expr>
(12) <id> ::= ID
```

Figure 3.3: The restricted CFG for ABL language.

3.2.2 The Exclusive Class Hierarchy

By following step two, an ECH in figure 3.4 can be constructed by hand or by a supporting tool. Each grammar symbol has a corresponding class, and the inheritance relations among these classes are derived from the productions of classification symbols. The directed arcs in the figure represent the inheritance relation based on productions (2), (4), (7), (10) and (12) in figure 3.3. Besides, *null* inheritance, nothing

inherited, is represented by '⊥' at the upper-right of the inheriting class.

3.2.3 Service Definitions

The third step is to define the services provided by symbols. Regarding name checking, <decl> (class) need have the following services: *declaration* returns the name and type of an identifier declared, *isReferenced* checks if the identifier is referenced, and *isRedeclared* checks if the name is duplicated. In ID and TYPE, *idName* and *typeName* provide the name and type of a declared variable accordingly. In <assign>, *reference* returns the names referenced, and *outReference* returns those referenced but not declared locally. Besides, <block> provides the followings: *symbol* returns a table of identifiers declared in its scope, *redeclared* checks if a given idName has other identical entries in the table, and *declared* checks if the idName is declared. In <id>, *isDeclared* checks if the identifier is declared. Other symbols are neglected because they provide no service.

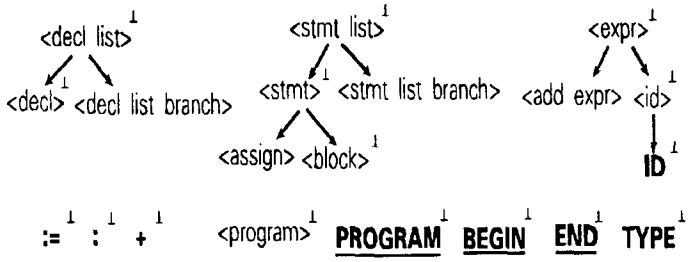

Figure 3.4: ECH for the restricted CFG of ABL language.

Through generalization, the common services of <assign> and <block>, *reference* and *outReference*, are moved to their superclass <stmt>.

The detailed definition for each service is specified based on pseudo sites and Smalltalk's syntax. For example, in <block>, service *symbol* returns a collection of declarations in site DeclNodes, all <decl> nodes in <block>, *reference* is the union of *outReferences* in site StmtNodes, all <stmt> nodes in <block>, *declared* recursively sends message to site Scope, the <block> node of the scope, and *redeclared* and *referenced* send messages *symbol* and *reference* to the <block> itself to check the circumstance of given idName. To simplify the description, the well known global objects in Smalltalk, such as classes of data structures, can also be used directly. For example, class *Association* and method *key:value:* are used in the definition of *declaration*. *includesKey:* and *includes:* have the same meanings as those in Smalltalk's *Dictionary* class. *ofKey:* retrieves all the elements matched in a collection. The service descriptions are shown in figure 3.6.

3.2.4 Pseudo Site Definitions

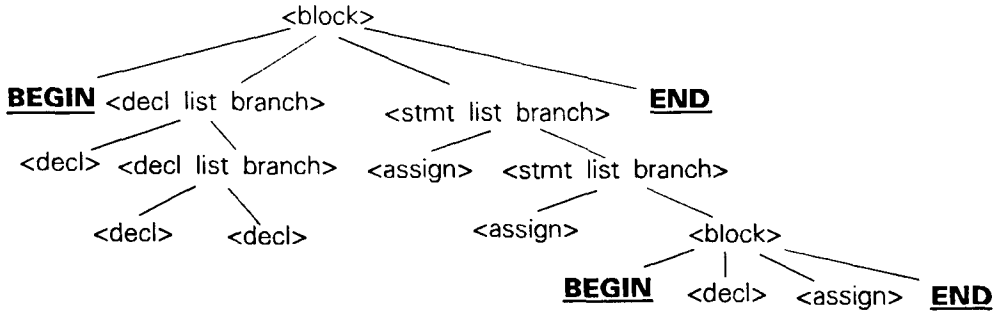

Figure 3.5 A structured parse tree of <block>.

The last step is to define pseudo site by path expression. For example, the pseudo site Scope used in service description of <block> represents the paths to nearest <block> ancestor. By drawing a parse tree on <block> as in figure 3.5, one can always find that there is a path from the lower <block> node to its nearest <block> ancestor. So, the path expression for Scope is "(<block>,-)*", any symbols but <block>, followed by (^<block>,-), the destination.

StmtNodes represents the paths to <stmt> offsprings. From the parse tree in figure 3.5, it is found that <stmt list branch> always repeats several times in a <stmt list>, concatenating a <stmt>,

<assign> or <block>, with a <stmt list>. So, StmtNodes is "<stmt list branch>* <stmt>". DeclNodes is similar to StmtNodes.

```
class <decl>    "<decl> ::= ID : TYPE "
Site
  Scope = (^<block>,-)* (<block>,-)
Protocol
  declaration
    ^ Association key: ID idName value: TYPE typeName.
  isReferenced
    ^ Scope referenced: ID idName
  isRedeclared
    ^ Scope redeclared: ID idName
class <stmt>    "<stmt> ::= <assign> | <block>"
Site
  DeclNodes = ∅; /* re-defined in subclass */
Protocol
  outReference
    ^ self reference - DeclNodes declaration.
class <assign>    "<assign> ::= <id> := <expr>"
Site
  IdNodes = any* <id>;
Protocol
  reference
    ^ IdNodes idName
class <block>
  "<block> ::= BEGIN <decl list> <stmt list> END"
Site
  StmtNodes = <stmt list branch>* <stmt>;
  DeclNodes = <decl list branch>* <decl>;
  Scope = (^<block>,-)* (<block>,-);
Protocol
  symbol
    ^ DeclNodes declaration.
  reference
    ^ StmtNodes outReference
  referenced: idName
    ^ self reference includes: idName
  redeclared: idName
    ^ (self symbol ofKey: idName) size > 1.
  declared: idName
    (self symbol includesKey: idName)
      ifTrue: [ ^ true ]
      ifFalse: [ ^ Scope declared: idName ].
class <id>    "<id> ::= ID"
Site
  Scope = (^<block>,-)* (<block>,-);
Protocol
  isDeclared
    ^ Scope declared: self idName.
```

Figure 3.6: The service descriptions and site definitions for ABL.

4. COMPILING PROCESS FOR A SAMPLE PROGRAM

The compiling process contains three phases: parsing, semantic analysis, and code generation. The parsing process is similar to traditional one except that the tree construction need be modified. A tree construction algorithm cooperating with traditional parsing actions will be presented in [11]. The tree constructed is a *structured* parse tree (SPT), in which each node has specific inherited properties. An

ECH can be simulated by a class hierarchy of single inheritance [11]. Semantic analysis and/or code generation can be done by extending topological sorting algorithm [4] on the *message dependency graph*, which represents the sending dependency on the messages of the nodes.

```
program OOP
  begin
    i : integer
    j : integer
    i := j + k
    begin
      k : real
      i := j + k
    end
  end
```

Figure 4.1: A sample program of ABL.

Here, we use a sample program of ABL language in figure 4.1, to show the compiling process of the model. The parse tree of the program is shown in figure 4.2. The structured parse tree is shown in figure 4.3. The dependency subgraphs for `<decl>`, `<assign>`, `<block>`, and `<id>` are shown in figures 4.5-4.7, respectively. The `ID` nodes in figure 4.7 are those which inherit the properties of `<id>`.

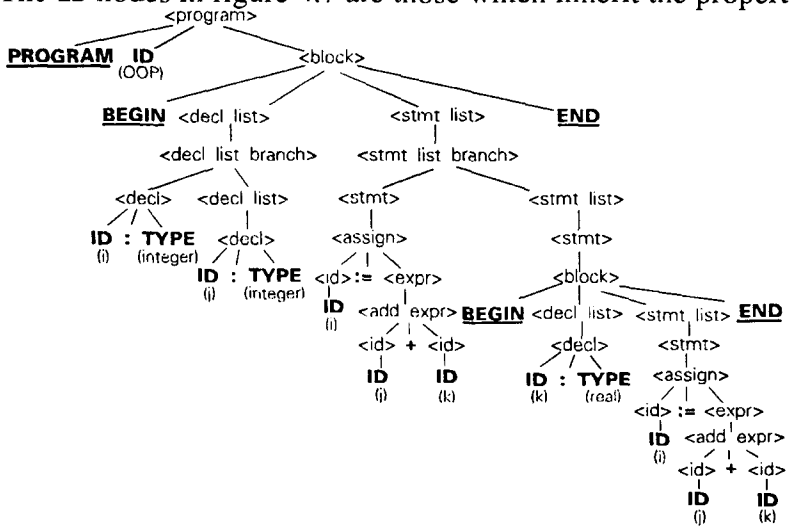

Figure 4.2: Parse Tree for sample program.

Some of the nodes are removed from the graph to make the presentation clear. The message names used in the graphs are the abbreviations of those in the previous section. We use the names `dcl` (`declaration`), `idn` (`idName`), `tpn` (`typeName`), `ref?` (`isReferenced`), `red?` (`isRedeclared`), `oref` (`outReference`), `ref` (`reference`), `sym` (`symbol`), `refed` (`referenced`), `red` (`redeclared`), `dcled` (`declared`), and `dcl?` (`isDeclared`).

5. COMPILER GENERATION

An OOCS can be used to generate the compiler too. Our research on the compiler generation is on going now, and several significant results have been achieved. Here, we introduce some of them roughly. The details can be found in [11] and [12].

The compiler based on OOCS is divided into three parts: parser, class hierarchy, and message evaluator. The restricted CFG of an OOCS is used to generate a parser and an ECH. The generation of parser is similar to that of traditional parser generator, e.g. YACC based on LALR(1), except that the parsing kernel is modified to construct an SPT. ECH can be simulated by a class hierarchy of an object-oriented programming language, or by a set of user defined types [11].

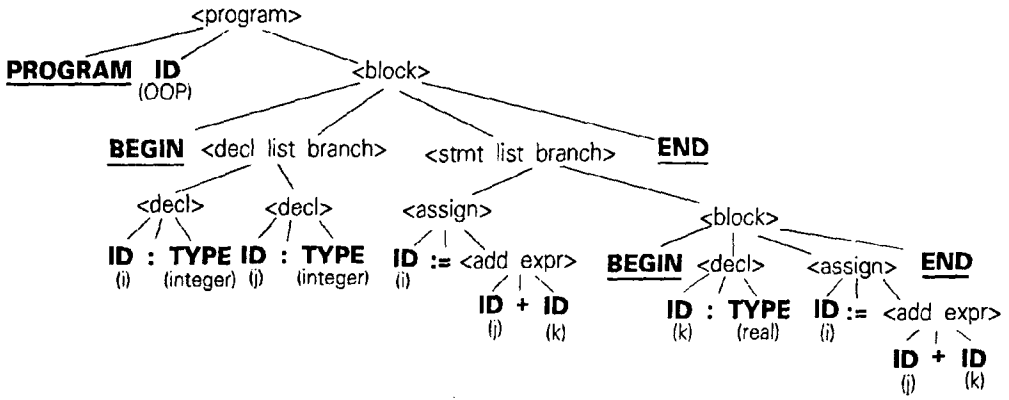

Figure 4.3: Structured Parse Tree for sample program.

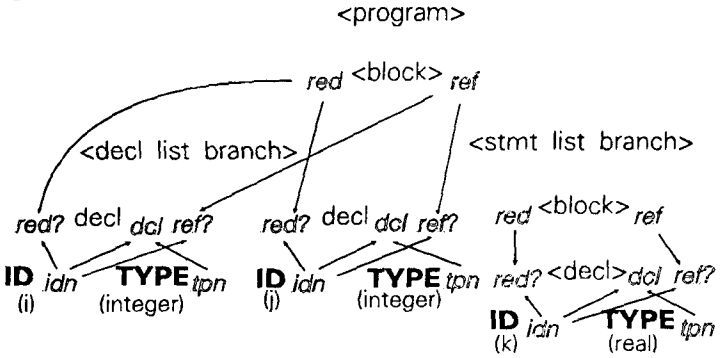

Figure 4.4: Dependency Subgraph of `<decl>`.

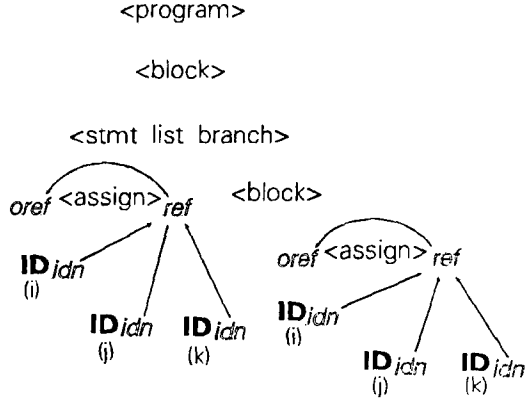

Figure 4.5: Dependency Subgraph of `<assign>`.

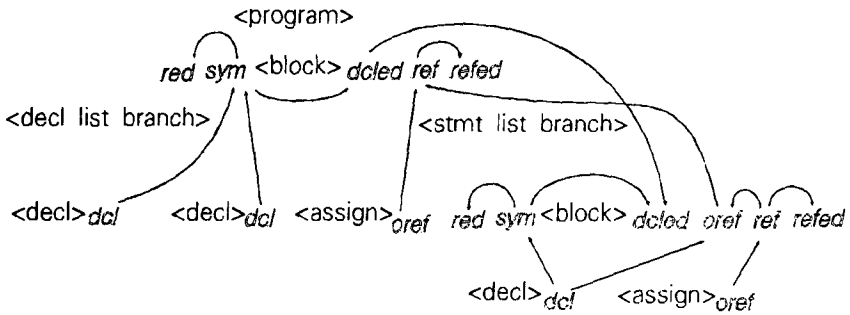

Figure 4.6: Dependency Subgraph of `<block>`.

The semantic analyzer and code generator are generated according to the service descriptions. Let's assume that the semantic analysis is a topological evaluation. A semantic analyzer generator need test whether a dependency graph could be circular or not first. By decorating each symbol with a set of dependency graphs and a set of DFA's states, it is shown that the OOCS contains circular dependency if and only if one such state patterns causes the graph to be circular. The process always terminate since the

patterns are finite. Thus, the circularity testing is decidable [12].

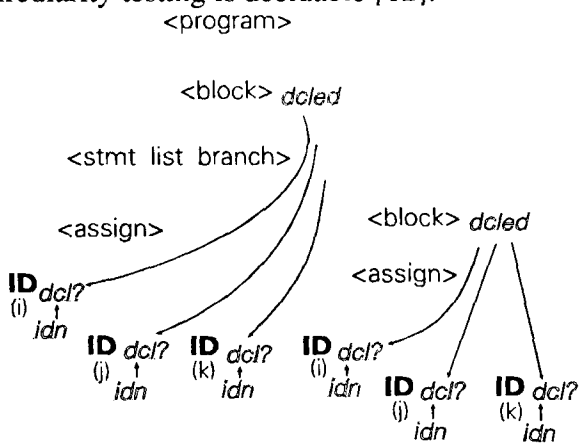

Figure 4.7: Dependency Subgraph of <id>.

6. CONCLUSION

We have presented an object-oriented approach to compiler specification. Our method treats grammar symbols as templates which instantiate nodes of parse tree for object-oriented semantics analysis. It uses restricted CFG to specify grammar rules for better classification and inheritance on semantics description. Based on a class hierarchy from the restricted CFG, an object-oriented analysis is specified by using path expression to describe possible receivers of messages. The use of path expression removes the restriction on the specification of message (value) propagations in attribute grammar methods.

Other interesting topics include 1) translating path expressions into node addresses of parse trees, 2) constructing a compiler generator based on the specification, and 3) applying this approach to language-based environment.

REFERENCES

1. Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, 1986.
2. Demers, A., Rogers, A., and Zadeck, F. K., "Attribute Propagation by Message Passing," In *Proc. of the 1985 Symp. on Language Issues in Programming Environments*, June 1985, pp. 43-59.
3. Hopcroft, J. E. and Ullman, J. D., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.
4. Knuth, D. E., "Semantics of Context-Free Languages," *Mathematical Systems Theory*, Vol. 2, No. 2, 1968, pp. 127-145.
5. Koskimies, K., "Software Engineering Aspects in Language Implementation," In *Proc. of the 2nd Compiler Compilers and High Speed Compilation Workshop, LNCS Vol. 371*, Spring-Verlag, 1989, pp. 39-51.
6. Nord, R. L. and Pfenning, F., "The Ergo Attribute System," In *Proc. of the ACM SIGSOFT'88 3rd Symp. on Software Development Environments*, 1988, pp. 110-120.
7. ParcPlace Systems, Inc., *Smalltalk-80 Version 2.5 Objectworks*, 1989.
8. Vorthmann, S. and Leblanc, R. J., "A Naming Specification Language for Syntax-Directed Editors," In *Proc. 1988 Conf. on Computer Languages*, Oct. 1988, pp. 250-257.
9. Watt, D. A., "Extended Attribute Grammars," *The Computer Journal*, Vol. 26, No. 2, 1983, pp. 142-153.
10. Wegner, P., "Dimensions of Object-Based Language Design," *ACM OOPSLA'87*, 1987, pp. 168-182.
11. Wu, P. -C. and Wang, F. -J., "Applying Classification and Inheritance to Compiling," submitted to *ACM OOPS Messenger*.
12. Wu, P. -C. and Wang, F. -J., "Message Passings on a Parse Tree," submitted to *19th Symp. on Principles of Programming Languages*.