编译实验三实验报告——Parser of PCAT

田应涛 10302010029　王曦　10300240014
陈济凡 10300240076　承沐南 10300240003

June 22, 2013

Contents

1 概述

本次 Project 包含五个部分

1. 抽象语法树 (Abstract Syntax Tree)
2. 类型检查 (Type Checking)
3. 栈帧 (Frames)
4. 中间表示 (Intermediate Representation)
5. 机器代码

本报告分为两大部分。前一大部分分别讲述 Project 所做的五个部分，后一大部分为实验结果。

2 抽象语法树 (Abstract Syntax Tree)

抽象语法树即使源代码的抽象语法结构的树状表现形式。树上的每个节点都表示源代码中的一种结构。之所以说语法是“抽象”的，是因为这里的语法并不会表示出真实语法中出现的每个细节。比如，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现；而类似于 if-condition-then 这样的条件跳转语句，可以使用带有两个分支的节点来表示。

我们实现的语法规则参照了 http://web.cecs.pdx.edu/apt/cs302_1999/pcat99/pcat99.html 中给出的 PCAT 标准, 并根据该标准写出了 EBNF 语法, 之后我们将该 EBNF 语法转换至 BNF 语法，即具体语法树。最后依据具体语法树制定了抽象语法树生成规则。

```
==========================
Abstruct Syntax - Modified
==========================

program                  -> ( Program ( body ) )
body                     -> ( BodyDef ( declarations-list statements-
    list ) )
declarations-list        -> ( DeclareList ( {declarations} ) )
declarations             -> ( VarDecs ( {var-dec} ) )
                         -> ( TypeDecs ( {type-dec} ) )
                         -> ( ProcDecs ( {proc-dec} ) )
var-dec                  -> ( VarDec ( ID type expression ) )
type-dec                 -> ( TypeDec ( ID type ) )
proc-dec                 -> ( ProcDec ( ID formal-param-list type body
    ) )
type                     -> ( NamedTyp ( ID ) )
                         -> ( ArrayTyp ( type ) )
                         -> ( RecordTyp ( component-list ) )
                         -> ( NoTyp () )
component-list           -> ( CompList ( { component } ) )
component                -> ( Comp ( ID type ) )
formal-param-list        -> ( FormalParamList ( {formal-param } ) )
formal-param             -> ( Param ( ID type ) )
statements-list          -> ( SeqSt ( { statements-list } )
```

```
statement               -> ( AssignSt ( lvalue expression ) )
                        -> ( CallSt ( ID expression-list ) )
                        -> ( ReadSt ( lvalue-list ) )
                        -> ( WriteSt  ( expression-list ) )
                        -> ( IfSt ( expression statement statement-
                            else ) )
                        -> ( WhileSt ( expression statement ) )
                        -> ( LoopSt ( statement ) )
                        -> ( ForSt ( ID expression-from expression-to
                            expression-by statement ) )
                        -> ( ExitSt () )
                        -> ( RetSt ( expression ) )
                        -> ( RetSt () )
                        -> ( SeqSt ( { statements-list } ) )
expression-list         -> ( ExprList ( { expression } ) )
expression              -> ( BinOpExp ( binop expression-left
    expression-right ) )
                        -> ( UnOpExp  ( unop expression ) )
                        -> ( LvalExp ( lvalue ) )
                        -> ( CallExp ( ID expression-list ) )
                        -> ( RecordExp ( ID record-init-list ) )
                        -> ( ArrayExp  ( ID array-init-list ) )
                        -> ( IntConst ( INTEGER ) )
                        -> ( RealConst ( REAL ) )
                        -> ( StringConst ( STRING ) )
record-init-list        -> ( RecordInitList ( { record-init } ) )
record-init             -> ( RecordInit ( ID expression ) )
array-init-list         -> ( ArrayInitList ( { array-init-list } ) )
array-init              -> ( ArrayInit ( expression-count expression-
    instance ) )
lvalue-list             -> ( LvalList ( { lvalue } ) )
lvalue                  -> ( Var ( ID ) )
                        -> ( ArrayDeref ( lvalue expression ) )
                        -> ( RecordDeref ( lvalue ID ) )
binop                   -> GT | LT | EQ | GE | LE | NE | PLUS | MINUS
    | TIMES | SLASH
                        -> DIV | MOD | AND | OR
unop                    -> UPLUS | UMINUS | NOT


==========================
Abstruct Syntax - Extension
==========================

in `get_comp_id`, # is ommited

program                 -> ( Program ( body #local-offset) )
proc-dec                -> ( ProcDec ( ID formal-param-list type body
    #level #local-offset ) )
var-dec                 -> ( VarDec ( ID type expression #level #
    offset ) )
formal-param            -> ( Param ( ID type #level #offset ) )
statement               ->
                        -> ( CallSt ( ID expression-list #type #level-
                            diff ) )
                        -> ( ForSt ( ID expression-from expression-to
                            expression-by statement #offset ) )
```

```
expression              -> ( BinOpExp ( binop expression-left
    expression-right #type #offset ) )
                        -> ( UnOpExp  ( unop expression #type #offset)
                             )
                        -> ( LvalExp ( lvalue #type #offset ) )
                        -> ( CallExp ( ID expression-list #type #level
                             -diff #offset) )
                        -> ( IntConst ( INTEGER #type ) )
                        -> ( RealConst ( REAL #type ) )
                        -> ( StringConst ( STRING #type ) )
lvalue                  -> ( Var ( ID type #level-diff #offset ) )
                        -> ( Var ( ID type) )   !! for TRUE/FALSE
```

3 类型检查 (Type Checking)

类型检查指的是对程序中的类型以及相关信息进行检查的一种程序分析过程。严格意义上的类型检查将会确保被检查的程序在执行时候不会有任何类型错误 (即程序是类型安全的)，但是更为通常意义上的类型检查提供了一定程度的类型安全性，但是不一定保证执行时候没有任何类型错误。

3.1 定义的类型检查

对于每个新的定义，我们将其加入当前空间之中，并检查是否有冲突。如果有的话，则需要报告错误。

```
VAR i := 10; (* OK )
VAR i := 20; (* Error: Name Conflict *)
```

鉴于 PCAT 要求在定义变量的时候赋上初始值，我们同样对此进行检查，要求初始值的类型必须不矛盾。

```
VAR i : INTEGER := 10;   (* OK *)
VAR i : INTEGER := 20.0; (* Error: Type Conflict *)
```