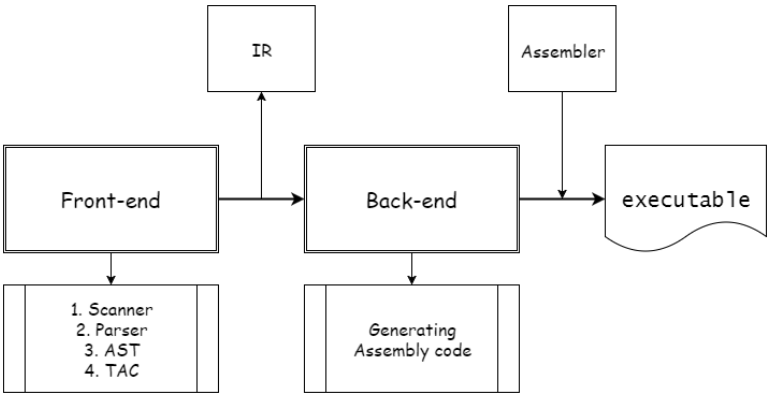
2013-11394 김형모
2013-11412 오평석

COMPILER

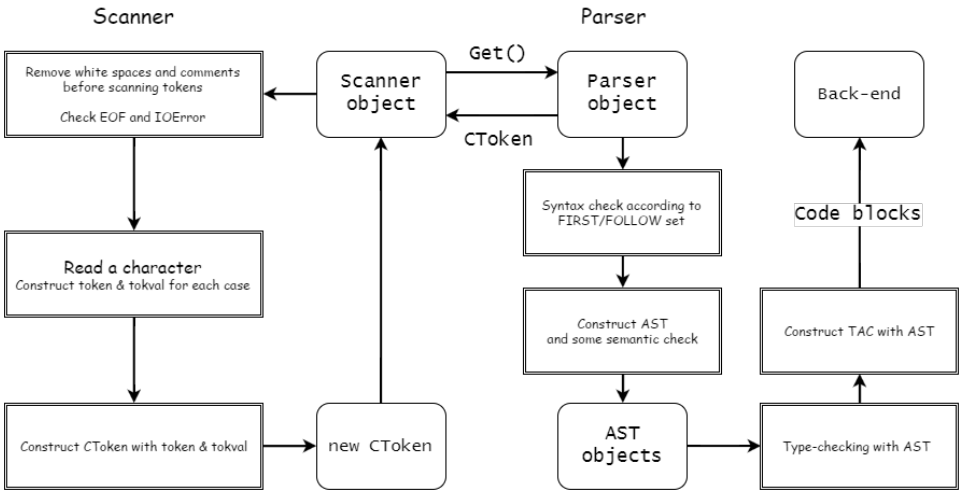
FINAL REPORT

1 Structure

In this project, we implemented SnuPL/1 compiler. It consists of front-end parts and back-end part. The front-end parts scan/parse given SnuPL/1 code and then, purform type-checking and make intermediate code. Because it supports no optimization features, back-end part just convert intermediate code into assembly code directly.



The scanner scans a character stream and outputs a stream of tokens. The parser uses these tokens to make AST, that is used to type-check and make IR: TAC symbol tables and code blocks. SnuPL/1 grammar is not pure LL(1), so we should have handle some ambiguous problems. Some semantic issues are checked when the parsing is going on, others are checked in type checking phase using constructed AST. The results(IR) are send to back-end. Next diagram shows these steps.



2 Front-end: Special issues

We handled special implementation issues as follows:

2.1 Scanner: Lexical analysis

The implementation of the scanner is somewhat straightforward process, but we should have config some policies for **excape sequences** and **character and string quotation errors**.

2.1.1 escape sequence

SnuPL/1 supports the following escape sequences : `\n`, `\t`, `\0`, `\‘`, `\‘`, `\\`. When the scanner faces `‘\’` character, it expects that the following character should generate escape sequence. Otherwise, it is considered as error.

For example, `\\` is `<tChar, ‘‘\\’’>` and `\a` is `<tUndefined, ‘‘\\a’’>`.

2.1.2 quotation error

Character: Stop when faced undefined situation.

1. Empty character.

   The scanner converts `‘’` into `<tUndefined, ‘‘\‘\‘’’>`.

2. Facing EOF.
   The scanner converts ‘ into <tUndefined, ‘‘\‘’’>.
3. Several characters. i.e. ‘ab..., ‘\nb..., etc.
   The scanner stops to build tokval until it realized that the input is undefined. It converts ‘ab into <tUndefined, ‘‘\‘a’’> and <tIdent, ‘‘b’’>. It converts ‘\nb into <tUndefined, ‘‘\‘\n’’> and <tIdent, ‘‘b’’>.
4. Invalid escape sequence. i.e. ‘\a.
   Handlie with invalid escape sequence is prior than stop to build tokval.
   It converts ‘\a into <tUndefined, ‘‘\‘\\a’’>.

String: For unclosed string, we have a little different policy.
The scanner recognizes the string token until the closing double quote, even if it classifies the token as tUndefined. If there is no closing double quote, the scanner add the character to string until EOF.

For example,

“Hello” <tString, ‘‘Hello’’>

“Hello <tUndefined, ‘‘\‘‘Hello’’>

“Hello\n” <tString, ‘‘Hello\n’’>

“Hello\a” <tUndefined, ‘‘\‘‘Hello\\a\’’’>

“Hello\a World” <tUndefined, ‘‘\‘‘Hello\\a World\’’’>

2.2 Parser: Syntax analysis

We constructed function table for FIRST and FOLLOW. These functions allow us to build a parsing table for SnuPL/1, which is important for making a predictive parser. We use **OCaml** language for generating follow entries.

2.2.1 FIRST/FOLLOW Set

Because the FIRST(assignment) and FIRST(subroutineCall) have ident together, we need to use more information. So, our parser is not pure LL(1) parser.

X	FIRST(X)
module	`module`
type	`boolean char string`
qualident	`ident`
factor	`ident number true false char string ( !`
term	`ident number,true false char string ( !`
simpleexpr	`ident number true false char string ( ! + -`
expression	`ident number true false char string ( ! + -`
assignment	`ident`
subroutineCall	`ident`
ifStatement	`if`
whileStatement	`while`
returnStatement	`return`
statement	`ident if while return`
statSequence	`ident if while return` ϵ
varDeclaration	`var` ϵ
varDeclSequence	`ident`
varDecl	`ident`
subroutineDecl	`procedure function`
procedureDecl	`procedure`
functionDecl	`function`
formalParam	`(`
subroutineBody	`var begin`

Also, because the FOLLOW(varDeclSequence) and FOLLOW(varDecl) have `;` together, we need to modify some productions to parse simply.

X	FOLLOW(X)
module	`EOF`
type	`) ; [`
qualident	`; , = # < <= > >= ) ] + - || * / && := else end`
factor	`; , = # < <= > >= ) ] + - || * / && else end`
term	`; , = # < <= > >= ) ] + - || else end`
simpleexpr	`; , = # < <= > >= ) ] else end`
expression	`; , ) ] else end`
assignment	`; else end`
subroutineCall	`; , = # < <= > >= ) ] + - || * / && else end`
ifStatement	`; else end`
whileStatement	`; else end`
returnStatement	`; else end`
statement	`; else end`
statSequence	`else end`
varDeclaration	`procedure function begin`
varDeclSequence	`; )`
varDecl	`; )`
subroutineDecl	`procedure function begin`
procedureDecl	`var begin`
functionDecl	`var begin`
formalParam	`: ;`
subroutineBody	`ident`

2.2.2 distinct subroutineCall & assignment from statement

When we look the EBNF syntax definition of SnuPL/1, we found interesting things. Look at the following production rules.

```
statement ::= assignment | subroutineCall | ...
assignment ::= qualidnet | ...
qualident ::= ident { ‘‘[‘‘ expression ‘‘]’’ }.
subroutineCall ::= ident ‘‘(‘‘ [ expression {‘‘,’’ expression} ] ‘‘)’’.
```

Ident is not only the element of FIRST(assignment), but also the element of FIRST(subroutineCall). Now suppose we face tIdent token when parsing

the statement. There is no way to distinguish between assignment and subroutineCall without viewing the next token.

To solve this, our parser check the symbol table when it faces that situation. If the symbol is not appeared in the symbol table, it is ‘‘`undeclared variable`’’ error. If the symbol type is procedure, the token belongs to subroutineCall. Otherwise, it belongs to assignment.

2.2.3 distinct varDeclSeqeuence & VaeDecl with semicolon

Similarly, we can not distinguish the semicolon in varDeclaration and varDeclSequence.

```
varDeclaration = [ ‘‘var’’ varDeclSequence’’ ‘‘;’’ ].
varDeclSequence = varDecl { ‘‘;’’ varDecl }.
```

Suppose we parse varDecl and face semicolon. Do we prepare to accept next varDecl, or to close varDeclaration? We don't distinguish two cases without checking the next token. So we modified EBNF syntax definition of SnuPL/1 by replacing varDeclSequence and adapted it to our parser.[1] Be careful about varDeclSequence is appeared not only in varDeclaration, but also in formalParam.

```
varDeclaration -> [ ‘‘var’’ varDecl { ‘‘;’’ varDecl } ‘‘;’’ ].
formalParam = ‘‘(‘‘ [ varDecl { ‘‘;’’ varDecl } ] ‘‘)’’.
```
[2]

[1] Two grammars are equivalent.

[2] varDeclSequence is replaced.

2.3 Parser and AST: Semantic analysis

In this phase, 1) we checks syntatical issues by consuming tokens and constructs a AST(Abstract syntax tree), and 2) we checks semantic issues by checking type of nodes in constructed parse tree recursively. Some semantic issues are checked before type checking.

- module/subroutine identifier match - parser.cpp
- duplicate subroutine/variable declaration - parser.cpp
- declaration before use - parser.cpp
- the types of the operands in expressions - ast.cpp
- the types of the LHS and RHS in assignments - ast.cpp
- the types of procedure/function arguments - ast.cpp
- the type of return value of function - parser.cpp(return array type) and ast.cpp(all other cases)
- the number of procedure/function arguments - parser.cpp
- catch open array when declaration - parser.cpp
- indexing with variable whose type is not an array - parser.cpp
- indexing with expression which is not a natural number - ast.cpp
- catch invalid constants - parser.cpp($> 2^{31}$) and ast.cpp($= 2^{31}$)

The last case may be somewhat different from other's implementation.

2.3.1 invalid number ($> 2^{31}$)

When constructing constant node, if number value is greater than 2^{31}, the parser raises error before type checking. If number value is exactly 2^{31}, checking boundary is postponed to typechecking phase.

2.3.2 invalid number ($= 2^{31}$)

We adopt 'simple' policy. If the number value is exactly 2^{31}, typechecking fails. But when unary '-' operation is attatched to number node directly, typechecking passes. For example, $-2147483648 + 1$ is correct input because opNeg is boxing 2^{31}. If unary '-' is not attatched directly to the number node, for example, $-2147483648 * 1$ is error case. This outputs same with reference parser (simple version)

Also, our compiler purform implicit type conversions for arrays and string-Constants in parser.cpp, using GetType() function.

2.4 AST: Intermediate code generation

In this phase of the project, we converted the abstract syntax tree into intermediate representation in three-address code form. The classes of AST nodes are : Scope, Statement, Expression. In general, scope has code symbols and temp symbols for statements. Each statement is interpreted into one(or more) three-address code. Each expression is translated to evaluated values. We may need temp symbols to use the value.

Array designators are also handled with expressions, but it has some issues including calcuating and calling pre-defined functions. Short-circuit codes are also difficult part of this phase. It requires generating many labels and gotos.

Lastly, we decided to merge 'opNeg' and integer constant even though our negation policy is simple.

2.4.1 Core implementation in expression's ToTac()

The most important parts in generating three-address-code are short circuit code and array designator.

opAnd

First, it gets the TAC of left operand by calling left->ToTac(cb, nextCond, ltrue). Next, it attaches the 'nextCond' label. Finally, it gets the TAC of right operand by calling right->ToTac(cb, ltrue, lfalse). It says that "If the current condtion is true, then look the next condition. Otherwise the total

condtion is false regardless of the remaining condition's result." It behaves as follows:

```
  if (cond1 is true) goto nextCond_1
  goto lfalse
nextCond_1:
  if (cond2 is true) goto nextCond_2
  goto lfalse
nextCond_2:
  ...
nextCond_n:
  if (condn is true) goto ltrue
  goto lfalse
ltrue:
  (some expression when the condition is true)
  goto next
lfalse:
  (some expression when the conditoin is false)
next:
```

opOr

It behaves similar when the operator is 'opAnd'. It says that "If the current condition is true, the total condition is true regardless of the remaining condition's result. Otherwise, then look the next condtion."

It behaves as follows:

```
  if (cond1 is true) goto ltrue
  if (cond2 is true) goto ltrue
  ...
  if (condn is true) goto ltrue
  goto lfalse
ltrue:
  (some expression when the condition is true)
  goto next
lfalse:
```

```
  (some expression when the condition is false)
next:
```

opNot

We create three labels with name "ltrue", "lfalse", and "lend". And we create the temporary variable (will be returned at the end) and construct the TAC of the control flow by calling ToTac(cb, ltrue, lfalse). Finally, we attaches the labels and add the 'goto' instruction to complete the control flow.

For example, when we face "a := !b" expression, our ToTac() function will works as follows:

```
  if (b is true) goto 1
  assign tmp <- true
  goto 2
1:
  assign tmp <- false
2:
  assign a <- tmp
```

CAstArrayDesignator

It calculates pointer of exact data and reference the pointer acourding to following steps.

First, we address pure array into pointer if needed. we should use pointer until return. Second, we calculate pointer to data location

The formular to evaluate location inside array as follows:

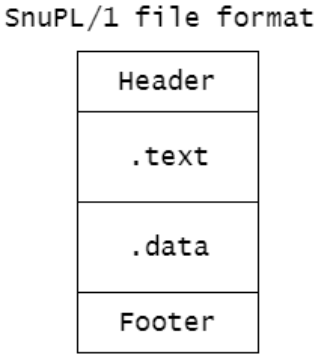
$$a[x1]...[xn] = (array\_offset)+(...((x1*C1+x2)*C2+x3)*C3+...+xn)*Cn$$

- &a = pointer to array 'a'
- array_offset = &a+ DOFS(&a)
- Ck = DIM(&a, k+1) when k = 1 to n-1
- Ck = data size of basetype

Last, we reference array location using CTacReference.

3 Back-end: Code generation

In this phase of the project, our compiler converts the TAC into x86 assembly code. Below diagram shows SnuPL/1 file format. The interesting parts of the file are .text section and .data section. .text section contains main and subroutines for the program. .data section contains space for global symbols.



3.1 Detailed implementation

3.1.1 EmitCode(), EmitScope(CScope *scope)

EmitCode() emits scope imformation of all subroutines include main(module).

EmitScope() emits everything related to a function. Theses constitute .text section.

1. Emits function name as a label. The label is used as access point in function call.
2. Computes stack offsets using symbol table.[3]
3. Sets up function prologue. Save old %ebp, set stack base, push callee saved registers, finally, make room for local variables.
4. Cleans up stack for local variable and setup meta data for local array.
5. Emits instructions in function body.
6. Release stack frame and callee saved registers.

[3] sum up all datasizes of local symbols, following alignment rules.

3.1.2 EmitData(), EmitGlobalData(CScope *scope)

EmitData() emits global data section with comments.

EmitGlobalData() emits labels, alignments, settings for global symbols that constitute .data section.

3.1.3 EmitInstruction(CTacInstr *i)

EmitInstruction() emits assembly code for each CTacInstruction.

binary operations

Convert tac 'dst = src1 **op** src2' into asm. Emit the following asm codes:

1. **load** src1 to **%eax**
2. **load** src2 to **%ebx**
3. operate **op %ebx**, **%eax**
4. **store %eax** to dst

unary operations

Convert tac 'dst **op** src' into asm. Emit the following asm codes:

1. **load** src to **%eax**
2. operate **op %eax**
3. **store %eax** to dst

memory operations

Convert tac 'dst = src' into asm. Emit the following asm codes:

1. **load** src to **%eax**
2. **store %eax** to dst

pointer operations

Convert tac 'dst = & src' into asm. Emit the following asm codes:

1. **load** the memory location of src to **%eax**
2. **store %eax** to dst

branch operations

For unconditional branching, convert tac '**goto** dst' into asm. Then emit: **jump** to **label** dst

For conditinal branching, convert tac '**if** src1 **relOp** src2 **then goto** dst' into asm. Emit the following asm codes:

1. **load** src1 to **%eax**
2. **load** src2 to **%ebx**
3. compare **%eax** and **%ebx**
4. conditional **jump** to **label** dst

function call-related operations

There are three special instructions for function call.

opCall: Convert tac '**call** dst <- src' into asm. Emit the following asm codes:

1. **call** src
2. remove local variable in stack frame (if num of params is 0 then skip this line)
3. **store %eax** to dst (if dst is null then skip this line)

opReturn: Convert tac '**return** src' into asm. Emit the following asm codes:

1. **load** src to **%eax**
2. **jump** to function's epilogue

opParam: Convert tac '**param** src' into asm. Emit the following asm codes:

1. **load** src to **%eax**

2. **push %eax** to stack frame

special operations
There are trivial instructions for:

opLabel: Emit asm **label:**

opNop: Emit asm **nop**

3.1.4 Operand and OperandSize

These are related with converting CTacAddr. Operand() returns an operand string for CTac.

CTacReference: stores offset(**%ebp**) to **%edi** and returns "(%edi)"

CTacName: returns "offset(%ebp)"

CTacConst: returns "$" + string of integer value

3.1.5 Operand and OperandSize

These are related with converting CTacAddr. OperandSize() computes the size of CTac.

CTacReference: reference can point only array, so brings basetype of innertype of pointer and return it's datasize

CTacName: returns type's datasize

CTacConst: returns 4 (Integer size)

With these substitution rules, we finally generated assembly codes from intermediate codes. We checked operations of real executable files with given and hand-made SnuPL/1 codes.